Data Integration Strategies for Distributed Reinforcement Learning in Robotics

Datenintegrationsstrategien für verteiltes verstärkendes Lernen in der Robotik

Wissenschaftliche Arbeit zur Erlangung des Grades
M.Sc. Maschinenwesen
an der Fakultät für Maschinenwesen der Technischen Universität München.

Themenstellender	Knoll, Alois Christian; Prof. Dr.-Ing. habil. Lehrstuhl für Robotik, Künstliche Intelligenz und Echtzeitsysteme
Betreuer	Walter, Florian; M.Sc. Lehrstuhl für Robotik, Künstliche Intelligenz und Echtzeitsysteme
Eingereicht von	Salcedo Bosch, Martí
Eingereicht am	02/12/2019 in Garching bei München

Data Integration Strategies for Distributed Reinforcement Learning in Robotics

Datenintegrationsstrategien für verteiltes verstärkendes Lernen in der Robotik

Wissenschaftliche Arbeit zur Erlangung des Grades
M.Sc. Maschinenwesen
an der Fakultät für Maschinenwesen der Technischen Universität München.

Themenstellender	Knoll, Alois Christian; Prof. Dr.-Ing. habil. Lehrstuhl für Robotik, Künstliche Intelligenz und Echtzeitsysteme
Betreuer	Walter, Florian; M.Sc. Lehrstuhl für Robotik, Künstliche Intelligenz und Echtzeitsysteme
Eingereicht von	Salcedo Bosch, Martí
Eingereicht am	02/12/2019 in Garching bei München

Ich versichere hiermit, dass ich die von mir eingereichte Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

München, 02/12/2019 Salcedo Bosch, Martí

Acknowledgments

I would first like to thank my thesis advisor M.Sc. Florian Walter of the Fakultät für Informatik at Technische Universität München for his support, advice and guidance along the whole project. He has always been available and he has steered me in the right direction when I needed it.

I would also like to thank my Munich friends for their support and help, they have been like a second family to me.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author

Martí Salcedo Bosch

Abstract

The field of reinforcement learning, developed during the nineteen-eighties and nineties, is a branch of machine learning which has consistently shown wide potential. Using this theory, it is possible to design computer programs able to learn which actions must be taken, in a given environment, to maximise a cumulative reward function. In other words, by rewarding the program, it is able to learn how to behave in order to solve a problem.

Originally this field was mainly applied to discrete and finite environments, however, it was possible to handle continuous environments using traditional function approximators. Recently the field has experienced a revolution, with the increase of the computational capacity, which enabled the use of artificial neural networks as function approximators. It has shown surprising results previously thought unfeasible and the number of fields where it may be applied has drastically increased. Robotics is one of them and in the past few years the achieved results have been very promising.

In general, and in robotics, one of the topics still to be deeply explored is the learning distribution. This distribution means to parallelise the learning, in other words, to have many workers facing the problem and sharing information instead of one isolated worker. With it, the learning can be optimised; involving shorter learning times and better knowledge of the environment among many other advantages. To contribute to this topic, in this project three different distributed architectures, based on the state-of-the-art algorithms, will be designed and implemented. The learning will be distributed using many simulated robotic arms, that will work in parallel performing the same task.

Contents

1. Introduction

This chapter introduces the lector to what motivated this project, the defined goals, scope and content.

1.1. Motivation

During the last five years the improvements on the field of reinforcement learning have been bigger than ever. The combination of the classical reinforcement learning algorithms with the power of the artificial neural networks has made this field very powerful and capable to solve some problems that had never been solved before. All indicates that this field will take a big part in the AI future, so the expectations are very high, but there is still a lot of work to be done.

Robotics is one of the fields where reinforcement learning has been applied. Nowadays the available robots, at least the industrial robots, mechanically are more than enough capable of anything. Otherwise, with regard to their intelligence the situation is still not very advanced. Basically industrial robots are directly programmed by humans who define exactly the movements that the robot will do. For instance, an industrial robot in a mounting chain at a car factory will perform exactly the same movements in a loop, which have been programmed before.

The main motivation in this work is to contribute in the combination of these two fields, robotics and reinforcement learning. Specifically focusing on the idea of the distribution of the learning among many robots. Enabling this way the possibility to use robots which work in different places to learn in parallel and to create a global knowledge. Always keeping in mind that these systems must be able to work in the real industry, where the deployed robots in the customer side must share the information without losing their confidentiality.

1.2. Goals

The main objective of this work is to design and implement one or more distributed reinforcement learning algorithms to be applied in Robotics. The algorithms must fulfil confidentiality requirements when sharing information and they have to be tested in a robotic environment, more precisely in a robotic arm simulator with the target of grasping objects.

To achieve this main goal, there are sub goals which have to be achieved before. It is necessary to acquire a deep understanding of the reinforcement learning theory, the classical and modern algorithms. The state of the art papers must be read and understood deeply in order to use their knowledge in this project. An enough Python knowledge with its AI libraries must be acquired to enable the implementation of the developed algorithms. Finally, a method to compare the results of the implemented algorithms must be created to asses them.

1.3. Scope

This project implements an existing deep reinforcement learning algorithm, which works with continuous action spaces. Afterwards, with the acquired knowledge, three distributed reinforcement learning architectures are conceptually developed and implemented in an actual program with a robot simulator. Then the results obtained are analysed and compared among the algorithms.

The work focuses on the distribution of the reinforcement learning, so this project does not go deep into the neural networks training, neither does into the computation efficiency. The same task is performed by the different distributed actors in the distributed algorithms. The algorithms are not tested in real physical robots. Not all the reinforcement learning parameters are deeply explored and tested, only the ones which have a strong effect on the distribution part of the algorithms.

1.4. Outline

The next chapter Background, describes the origin of the reinforcement learning theory and how it has evolved over the last forty years, from the tabular algorithms to the modern algorithms, which use neural networks as function approximators. The last part of the chapter introduces the distributed reinforcement learning and which are its most notorious and state of the art algorithms. The following chapter Algorithms,

the main part of the thesis, explains the distributed reinforcement learning algorithms developed in this work. The next chapter Implementation describes how the algorithms have been implemented. Subsequently the Results chapter shows the results of the experiments and how they have been obtained, then they are compared to assess the algorithms performance. Afterwards, the chapter Conclusions analyses all the results and summarises the main points that can be drawn from them. Finally the last chapter Future work contains some ideas that have not been implemented in this work, due to the lack of time, but are considered interesting for a further investigation.

2. Background

This chapter contains a introduction to reinforcement learning and how this field has evolved to the most recent algorithms.

2.1. Reinforcement learning

Reinforcement learning, as one of three machine learning paradigms, was born in the last twenty years of the twentieth century. Richard S. Sutton is considered one the founding fathers of this theory. His book "Reinforcement Learning: An Introduction"[17] published in 1998, written with Andrew G. Barto is the best introduction to this field and most of the new theories are based on its methods.

The RL (Reinforcement Learning) theory is based on the idea of making machines learn like a complex living being would do, learning from the experience acquired after exploring its environment and getting different rewards depending on the taken actions. For example, a child learning to walk. The learning process is based on trial/error and cause/effect, every time the child falls a negative reward is given (pain) which will affect the future actions. The more experiences the child gets the better the walking is performed, and finally after many experiences the child is able to walk.

To formalise the previous idea of RL five elements are defined:

- ***Agent*** Learner and decision maker, the child in the previous example.
- ***Environment*** Everything that is not the *Agent*, the physical world in the previous example.
- ***Reward*** Function that defines a numerical reward obtained for each reached state.
- ***Policy*** Function to map from state to action. Defines which action must be taken depending on the current state. Finding this function is the actual learning.
- ***Value function*** Defines how valuable a state is. It is the sum of the expected future rewards when starting from a state following a certain policy. Alternatively it can be expressed as an ***Action-value function*** which defines how valuable is to take an action in a certain state. It is the sum of the expected future rewards when an action is taken in a state following a certain policy.

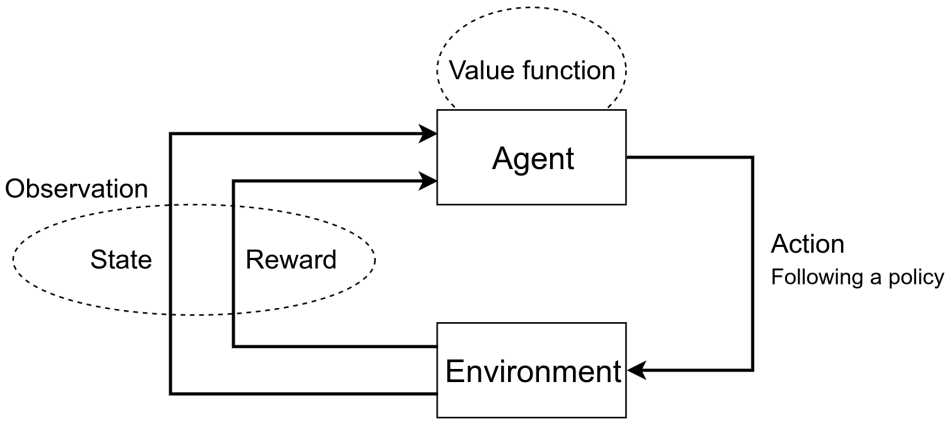


Figure 2.1.: Agent–Environment interaction.

The first two elements interact in a closed loop, shown in the Figure 2.1. The *Agent* observes the *Environment* to know its current state and based on that decides which action to take, which will lead to a new state and will produce a reward after the interaction with the *Environment*. Based on that, Sutton models mathematically the RL as a discrete MDP (Markov Decision Process). The MDP has a discrete number of time time steps $t = 0, 1, 2, 3, \ldots$. At each time step the *Agent* observes the *Environment* to know its current *state* $s_t \in \mathcal{S}$, based on the state and following a certain *policy* $\pi(a|s)$ an *action* $a_t \in \mathcal{A}(s)$ is taken. The action produces an interaction with the *Envirnoment* generating a *reward* $r_t \in \mathcal{R} \subset \mathbb{R}$ and leading to a *new state* s_{t+1} based on the probability $p(s_{t+1}|s_t, a_t)$. This pattern in a closed loop creates a trajectory $s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots$

The main goal in RL is to find a *policy* $\pi(a|s)$, which given a *state* s_t tells what *action* a_t has the biggest probability to maximise the sum of the future *rewards* $r_{0,1,2,\ldots}$. In most of the real RL applications the number of steps is unknown or it could be infinite in control tasks. Hence, the sum of the future *rewards* could be infinite:

$$R_t = \sum_{k}^{\infty} r_{t+k} = \infty \tag{2.1}$$

To solve this issue a *discount rate* $\gamma \in [0, 1]$ is defined, which defines how far in the future the algorithm looks. With this rate γ the sum of the future *rewards* is defined as:

$$R_t = \sum_{k}^{\infty} \gamma^k r_{t+k} \tag{2.2}$$

This new representation (Equation 2.2) allows to write the sum of the future rewards as recursive expression:

$$R_t = r_t + \gamma R_{t+1} \tag{2.3}$$

Based on this last formula (Equation 2.3), the *value function* and the *action-value function* can be easily defined as Bellman equations.

- The *value function* is defined as the expected sum of the future *rewards* staring from a *state* s_t following a policy π:

$$v_\pi(s) = \mathbb{E}_\pi[r_t + \gamma v_\pi(s_{t+1})|s_t = s] \tag{2.4}$$

- The *action-value function* is defined as the expected sum of the future *rewards* staring from a *state* s_t and taking an *action* a_t following a policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[r_t + \gamma q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a] \tag{2.5}$$

Since a *value function* is determined by a policy the following expression is satisfied:

$$\pi \geq \pi' \iff v_\pi \geq v_{\pi'} \tag{2.6}$$

Thus there is always at least one policy which is better than the others, called *optimal policy* $\pi^*(a|s)$. Knowing the value functions (Equation 2.4 and Equation 2.5), if a maximum *reward* is wanted, the *policy* must be defined to choose the actions that lead to a state with the highest value function among the other states possible. Using this criteria an *optimal policy* π^* is followed. This kind of policy is called *greedy policy*.

In the real problems, initially the *value functions* are unknown, so the *environment* has to be explored going through the different *states* to update the *value functions*. The better the *value functions* the better the result of the algorithm following an *optimal policy* π^*. To explore the *environment* as much as possible while training the *value functions*, it is not recommended to take always *actions* which lead to the most valuable *states*. Usually an exploration rate $\varepsilon \in [0, 1]$ is defined, which determinates the chance to take a random action in order to explore new possible states. This training strategy is called $\epsilon - greedy$ and is one of the most extended in the RL algorithms.

Using these principals, the first RL methods were defined (first part of Sutton's book [17]). The first method is called Dynamic Programming which is able to learn an optimum policy but requires a model of the system (which in most of the cases is unknown). The second method is called Monte Carlo Methods which does not need a model of the system, and therefore has a wider range of applications, but requires

reaching the final state of an episode to perform the learning, is to say, it is necessary to go through all the states until the last one to get the final outcome. Finally, a third method was developed, called Temporal Difference. This method combines advantages of both previous algorithms. No model of the system is required and after every action the agent is able to learn without the need to wait until a final state. This method is the one used in the most famous classical RL algorithms, such as SARSA or Q-Learning.

The Q-Learning algorithm (Watkins, 1989)[18] is the basis for the algorithms used in this work, therefore it is completely developed in the Algorithm 1 to facilitate a quick access to the lector.

Algorithm 1 Q-Learning

Define the algorithm parameters: learning rate $\alpha \in (0, 1]$, exploration rate $\epsilon \in [0, 1]$ and number of episodes to be run E
Initialise $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily except that $Q(S_{terminal}, \cdot) = 0$
for $episode = 0$ **to** E **do**
  Initialise states S
  for each *step* **in** *episode* **do**
    Choose action a_t which maximises Q with a chance ϵ of a random action
    Take action a_t, observe reward r_t and next state s_{t+1}
    Update $Q(s_t, a_t)$:
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
    Next step becomes current step $s_t \leftarrow s_{t+1}$
  end for
end for

2.2. Deep reinforcement learning

The previously described classical Temporal Difference algorithms, such as Q-Learning, have a wide range of applications. They can be combined with function approximators (second part of Sutton's book [17]) to work with continuous state spaces, and even in continuous action spaces, which is required to apply these algorithms into fields such as robotics. Originally traditional function approximators, such as interpolators, were used, but when the computational capacity grew it was possible to replace them with ANN (Artificial Neural Networks). For example in Q-Learning algorithm the $Q(s, a)$ function, which usually is defined by a table, would be replaced by an ANN $Q(s, a|\theta)$. The combination of this two fields, ANN and RL, was a breakthrough. The algorithms which use this combinations are called deep RL in this work.

In 2013 a paper called "Playing Atari with Deep Reinforcement Learning"[9] was published by the company Deep Mind. It was the first successful attempt to learn directly from a very high dimensional space, such as raw pixels of an image, using a modified version of the Q-Learning algorithm which uses an ANN as Q and a memory to store the transitions data in order to train the ANN Q without producing learning instabilities. This new training method was called "memory replay". So instead of learning directly from each transition (s_t, a_t, r_t, s_{t+1}) when it takes place, the transitions are stored in a memory and the Q network is trained picking a defined number of random transitions from this memory (mini batch) in each step of the episode.

Two years later, in 2015, they published another paper, called "Human-level control through deep reinforcement learning"[10]. This was a new version of the algorithm which added a new network Q', called "target network", to increase even more the stability. This so called "target network" is a copy of the Q network. It is used to update the Q modifying the Bellman equation in the following way: $Q(s_t, a|\theta) = r_t + \gamma max_a Q'(s_{t+1}, a|\theta')$ So instead of using the same Q network to update itself, the target network Q' is used. Then the weights of Q' are updated every certain number of episodes with the weights of the original Q network. This algorithm was called DQN (Deep Q-Network) and was a real revolution.

After these papers, the use of these two new techniques, "memory replay" and "target network", have been applied to many other algorithms. This caused a revolution on the field and new papers with improvements are published almost every month.

One of the big handicaps of the DQN algorithm is that it can only be applied to discrete action spaces, is to say, the algorithm defines which action to take among a list

of possible actions $a^0, a^1, a^2, \ldots, a^n \in \mathcal{A}$. On the other hand, an algorithm which works with continuous action space, defines a $value \in [min, max] \subset \mathbb{R}$ which each action has to take. The discrete action space constrains notoriously the fields where DQN can be used. For example, a robotic arm, which needs real values as commands to move its joints. This was solved in the paper called "Continuous control with deep reinforcement learning"[7] which was published in 2016. The paper applies these two new techniques to a Deterministic Policy Gradient allowing it to work with ANN, this way a policy network called "actor" μ is used to map states to a continuous action space. To train the actor, another network is used, the "critic" network Q. This network maps states and actions to a real value, which defines how good is the taken action at the given state (The same function as the original Q function), so basically tells how good the actor network generated the actions, that's why it is called critic. The critic network is updated using the methods mentioned in the previous papers [9][10]. Otherwise, the actor uses the critic to asses how good the output action was at the given input state. Then the actor weights are updated modifying the output action in order to maximise the output value of the critic. (Deeply explained in chapter 3) This new algorithm called DDPG (Deep deterministic policy gradient) extends the RL to problems where the DQN was not possible to be applied.

Thanks to DDPG algorithms or similar, the deep RL applied to robotics has also experienced marked improvement. Papers such as "Comparing Task Simplifications to Learn Closed-Loop Object Picking Using Deep Reinforcement Learning"[2] published this year (2019), show very successful results of deep RL algorithms applied to real robots performing grasping tasks. In parallel, the advantages of the simulated robots, since they can be trained easily and faster, have encouraged the development of simulated robot platforms which allow deep RL algorithms to be trained. One example is the OpenAI robotics environments based on MuJoCo physics simulator, the one used in this work. This environments are described in the paper "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research"[13] published last year (2018).

2.3. Distributed reinforcement learning

In parallel to the robotics development, there is another branch of deep RL which has been advancing the last few years: the subfield called Distributed RL, which aims to distribute the learning among many agents in order to increase the learning speed and to reach a wider knowledge of the environment.

One of the oldest architectures found in this field is the one called "Gorila", described in the paper "Massively Parallel Methods for Deep Reinforcement Learning."[11]. It describes a distributed learning based on many agents which acquire the experiences and share them with multiple learners. These learners take some part in the learning and calculate the gradients. Afterwards the gradients are sent to a parameter server where the actual networks are updated. Subsequently the parameter server sends the updated weights to all the actors and learners to update them and close the loop.

After "Gorila", one of the most successful architectures was developed, the A3C, described in the paper "Asynchronous Methods for Deep Reinforcement Learning"[8] published in 2016. The main idea behind A3C is to run the episodes in parallel using multiple actors in different threads. This way the experiences are gathered much faster and with a wider range on differences, which leads to a better knowledge of the environment. All the actors have access to a global network which is trained by all of them simultaneously. A3C has set a benchmark for further distributed RL development. A good example of this further development is the architecture called IMPALA described in the paper "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures"[4] published in 2018.

One of the architectures focused specially in distributed DDPG algorithms is the one called "D4PG", described in the paper "Distributed Distributional Deterministic Policy Gradients."[1]. Basically this architecture spawn many actors which gather transitional experiences (s_t, a_t, r_t, s_{t+1}) and send them to a central learner which is trained using them.

Some of these architectures have already been applied in robotics. The paper "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation."[14] shows a distributed architecture, very similar to the A3C one, with a continuous action space using a DDPG algorithm. Another example, the paper "Distributed Reinforcement Learning for Multi-robot Decentralized Collective Construction."[15] also describes an architecture based on A3C with multiple real robots applied to construction.

Having reached this point, one wonders what should be next. The Distributed RL applied to robotics is still to be deeply explored, that is why this project will try to go further on the matter and see if improvements with respect to the current methods can be achieved.

This project focuses specially on, above described, distributed algorithms. To begin this exploration, the existing algorithms have been classified based on some defined parameters. Subsequently the algorithms have been pictured in a combinational tree to enable a visual analysis. This analysis will help to clarify what has already been done and what could still be explored in the distributed RL.

To define the parameters for the analysis, some of the most known papers, related to the field, have been used [8][11][4][1]. The parameters have been extracted after comparing the architectures defined in these papers, discarding all the other aspects not related to distributed RL. The Table 2.1 shows the parameters obtained after the analysis. Of course this classification has a non-unique solution, so more parameters could be defined or the ones described here could be seen from a different point of view.

The Figure 2.2 pictures the parameters in a combinational tree. As it can be seen, the last parameter "Actors tasks" is not used in the tree, because in this work the defined algorithms learn only one task and also because all the classified existing algorithms focus on one task at a time too. Impala is the only one tested in different tasks, but this is omitted in the tree. Learning multiple tasks simultaneously will be considered as part of the possible future research, described in the chapter Future work.

Note: it is important to differentiate the Distributed RL from the Federated RL because their similarity could lead to misunderstandings. The papers "Federated Reinforcement Learning."[20] and "Federated Machine Learning: Concept and Applications"[19] describe the so called Federated RL and will help the lector to get a better understanding of such field.

Learning in the actor	
Actor learning	Part of the learning takes place in the actors.
Non-actor learning	The actors do no take part in the learning.
Number of learners	
One learner	There is only one learner.
Multiple learners	The learning is distributed or parallelised among many learners.
Acting-learning bound	
Decoupled	The acting and the learning are done in parallel and with independence.
Coupled	The acting steps are coordinated with the learning procedure.
Experience storage	
Actor storage	Each actor storages its own experiences.
Global storage	The experiences are stored in a global memory.
Actors synchronisation	
Continuous	The actors and the learners share and use the same networks, so there is no need for synchronisation.
Discrete	The actors and the learners use different networks. They must be synchronised periodically.
Actors tasks	
Same task	Each actor performs the same task with different random conditions.
Different tasks	Each actor or group of actors performs different tasks to learn them simultaneously.

Table 2.1.: Distributed RL classification parameters

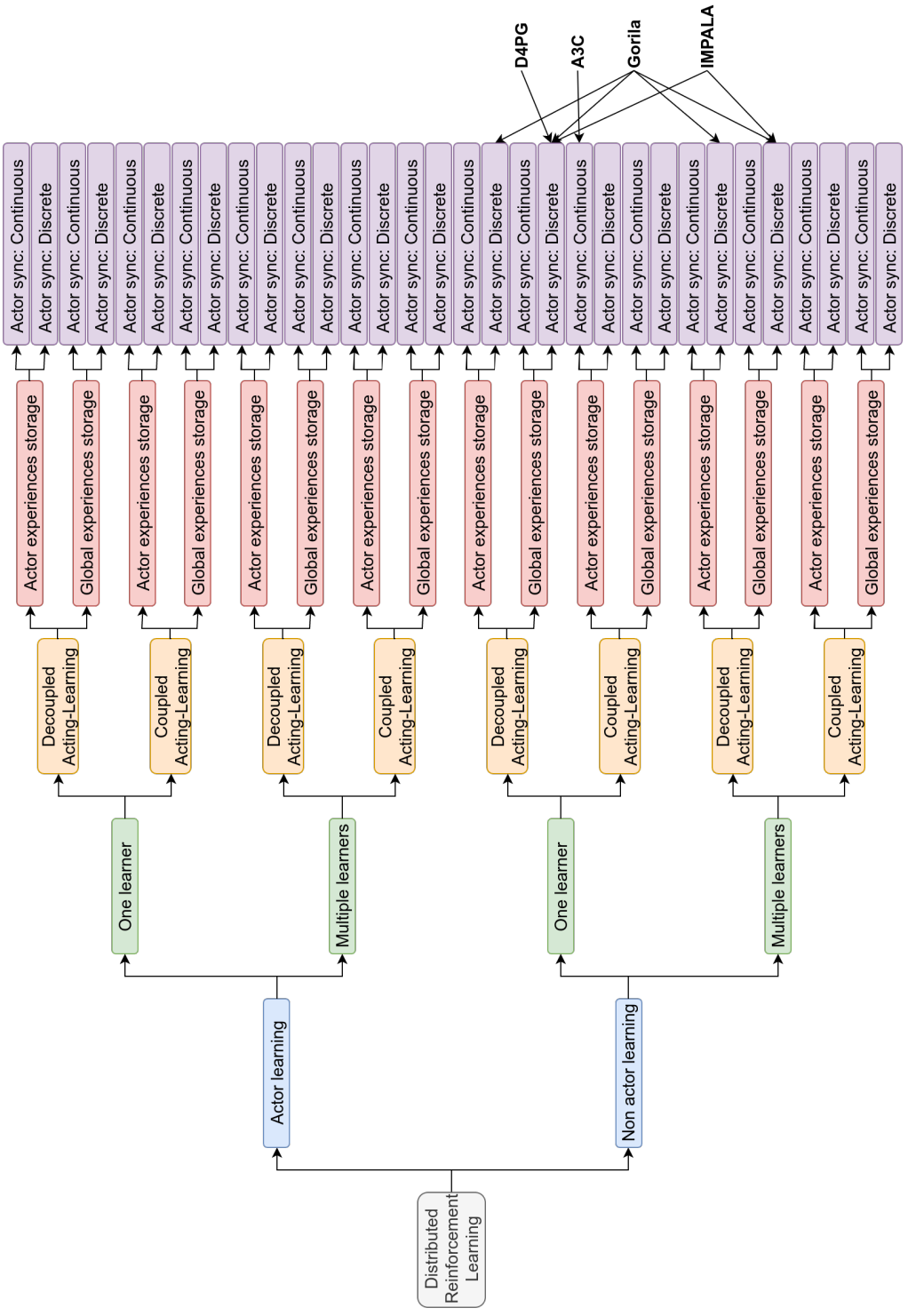


Figure 2.2.: Classification tree of the distributed algorithms

3. Algorithms

After the study of the background in the previous chapter, and following the defined goals, three architectures have been developed and implemented. The first architecture is a distributed algorithm with multiple actors and one unique central learner, this architecture has been designed in order to simulate different physical machines, so the actors and the learner must send the data to each other. The second architecture is an attempt to explore one of the unexplored branches in the classification tree (Figure 2.2). This architecture has multiple actors without a central learner. As the previous architecture, this one simulates different physical machines, so the actors must send data to each other. Finally, the third architecture is the same as the first one, with the difference that it has been designed so the actors and the learner work in the same machine with a shared memory space, this way all the actors have immediate access to the learner and vice versa, so there is no need to send any data. This last architecture has been implemented in order be able to asses the effect of the communications in the performance.

The RL algorithm used in the three architectures is the DDPG (Deep deterministic policy gradient) [7]. This algorithm has been chosen because it is able to work in a continuous action space, and this is needed in this project since the robotics simulator used works with a continuous action space. Besides, most of the distributed architectures, documented in the papers, are based on discrete action space algorithms, so implementing the distributed architecture with a continuous action space has been considered as a difficulty plus.

3.1. Deep Deterministic Policy Gradient

The algorithm DDPG (Algorithm 2), described in the paper "Continuous control with deep reinforcement learning"[7], has a similar structure to the Q-Learning (Algorithm 1) and specially to the DQN[10], since it uses the two methods explained in the section 2.2, "memory replay" and "target network". Hereunder the DDPG is deeply explained and it is written in pseudocode on the Algorithm 2.

First some hyper parameters must be defined:

- Maximum memory capacity $C \in [1, \infty) \subset \mathbb{N}$
- Initial exploration rate $\epsilon \in [0, 1] \subset \mathbb{R}$
- Minimum exploration rate $\epsilon_{min} \in [0, \epsilon] \subset \mathbb{R}$
- Exploration rate decay $\delta \in [0, 1] \subset \mathbb{R}$
- Mini batch size $N \in [1, C] \subset \mathbb{N}$
- Discount rate $\gamma \in [0, 1] \subset \mathbb{R}$
- Learning rates $\eta_{critic}, \eta_{actor} \in [0, \infty) \subset \mathbb{R}$
- Target update rate $\tau \in [0, 1] \subset \mathbb{R}$

Initially the model networks are created, the critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with random weights θ^Q and θ^μ. Then the target networks critic Q' and actor μ' are created, with the same weights as the model networks, so the model and the target networks are initialised identical. After that, the replay memory R is initialised empty with a maximum capacity C. Now the main loop may begin. This loop runs an episode in each iteration in order to do the training. Before every episode begins, the environment is reset to start with random initial conditions, and to obtain the initial state $s_{t=0}$. Then the episode begins here with the second loop, each iteration in this loop is a step of the episode. For each step in the episode:
With probability ϵ a random action is generated, following a probabilistic distribution, or otherwise the action is obtained with the model actor network:

$$a_t = \mu(s_t|\theta^\mu) \tag{3.1}$$

Then if the exploration rate ϵ is bigger than the minimum exploration rate ϵ_{min}, the exploration rate ϵ is decreased using the exploration rate decay:

$$\epsilon \leftarrow \delta\epsilon \tag{3.2}$$

After that the action a_t previously obtained is executed in the environment and the reward r_t and the next state s_{t+1} are obtained. Consecutively this transition (s_t, a_t, r_t, s_{t+1})

is stored in the memory R. If the memory has reached the maximum capacity the oldest transition is erased and this new one is added.
Now here begins the training part. Randomly N transitions (s_i, a_i, r_i, s_{i+1}) from the R memory are sampled. For each of these transitions the target values for the critic network Q are calculated. These target values, called y_i, are calculated using the Bellman equation with the target critic and actor networks:

$$y_i = \begin{cases} r_i & \text{if } s_{i+1} \text{ is } \textit{terminal} \\ r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) & \text{otherwise} \end{cases} \tag{3.3}$$

Then the squared difference of these values y_i with the current values $Q(s_i, a_i|\theta^Q)$ is calculated, and afterwards arithmetically averaged, in order to have a differentiable loss function with respect to the model critic network weights θ^Q.

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(s_i, a_i|\theta^Q))^2 \tag{3.4}$$

Consecutively the model critic network Q is updated trying to minimise the previous calculated loss function, so the difference between the target values y_i and the current values $Q(s_i, a_i|\theta^Q)$ is reduced. There are no target actions to train the actor value. The only way to know how good was an action a_t in a state s_t is using the critic network Q. The problem is that the critic network Q is not differentiable with respect the actor network weights θ^μ, thus it is differentiated with respect to the actions a and subsequently, using the actor network μ with the chain rule, differentiated with respect to the actor weights θ^μ. Unlike the previous case, here the weights are updated maximising the function, since a high Q value is wanted. This is done through all the samples and later arithmetically averaged to update to the network.

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i} \tag{3.5}$$

After the training of the model networks, the target networks are updated using τ to get them closer to the model networks progressively.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \tag{3.6}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \tag{3.7}$$

Finally, if the step s_{t+1} is a goal step, which means that the goal of the task has been achieved, the episode loop is broken and a new episode must begin. If it is not, then the step s_{t+1} becomes the current step s_t and a new iteration of the episode starts.

Algorithm 2 Deep Deterministic Policy Gradient (DDPG)

Initialise critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with random weights θ^Q and θ^μ
Initialise target critic Q' and actor μ' with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialise replay memory R with capacity C
for $episode = 1$ **to** M **do**
  Initialise environment and get initial state s_1
  for $t = 1$ **to** T **do**
    With probability ϵ select a random action a_t or elsewhere $a_t = \mu(s_t|\theta^\mu)$
    if $\epsilon > \epsilon_{min}$ **then**
      Update exploration rate with δ:
      $\epsilon \leftarrow \delta\epsilon$
    end if
    Execute action a_t and obtain reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in memory R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Calculate Bellman equation:
    if s_{i+1} is *terminal* **then**
      $y_i = r_i$
    else
      $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    end if
    Update critic by minimizing the loss with learning rate η_{critic}:

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(s_i, a_i|\theta^Q))^2$$

    Update actor using the sampled policy gradient with learning rate η_{actor}:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i}$$

    Update the target networks with τ:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

    if s_{t+1} is a *goal_state* **then**
      break loop to finish the current episode
    end if
    Next state becomes the current state $s_t \leftarrow s_{t+1}$
  end for
end for

3.2. Distributed architecture with centralised learning

As described at the beginning of the chapter 3, this is a distributed architecture with many actors and one unique central learner. It has been designed in order to simulate different physical machines, so the actors must send the data to the learner and vice versa. The communication is based on memory containers which are accessible by the actors and the learner. Periodically the actors send gradients, generated with mini batches of their own memory, and the learner takes them from the containers to train its networks. The other way round, the learner sends a copy of its weights to the actors so they update their networks. Sending gradients and weights the confidentiality is fulfilled, which was one of the goals of the project. The Figure 3.1 shows a schematic of the architecture.

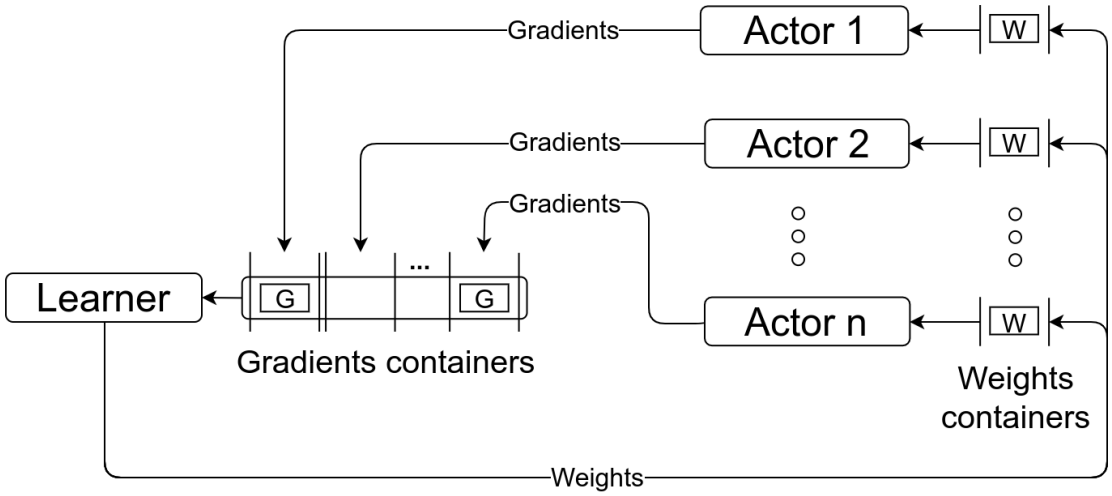


Figure 3.1.: Distributed architecture with centralised learning.

In order to clarify this distributed architecture and to be able to compare it with the other existing algorithms, it is classified with the parameters Table 2.1:

- **Learning in the actor:** Non-actor learning
- **Number of learners:** One learner
- **Acting-learning:** Decoupled
- **Experience:** Actor storage
- **Actors sync:** Discrete

After this analysis, and comparing the parameters with the ones in the Figure 2.2, it is possible to see that this architecture can be classified as one the possible architectures defined in the paper "Gorila"[11]. The pairs actor-learner in the "Gorila" architecture here would be the actors, and the so called parameter-server in "Gorila" here would be the central Learner. So the system described here is very similar to one of the possible architectures defined in "Gorila", but with the main difference that here DDPG is used instead of DQN[10] as the learning algorithm.

The architecture is divided in two parts, the learner and the actors. The algorithms written in pseudocode are described on Algorithm 3 and 4. Since the learning algorithm is the DDPG, and it has already been described, the explanation of the algorithms here will focus on the distributed architecture, so the training parts will be omitted, but the parts which differ a lot from the DDPG and therefore require clarification.

The actors:

At the beginning each actor n initialises its own critic and actor networks, model Q^n, μ^n and target Q'^n, μ'^n. Then the actor waits while its *weights_container*^{n} is empty. When the *weights_container*^{n} is full, which means that the learner has sent a copy of its initial weights to the actor n, the learner weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ are "taken" from the container. ("Taken" means that the weights are extracted from the container and the container is emptied.) The learner weights are copied to the actor weights $\theta^{Q,n}, \theta^{Q',n}, \theta^{\mu,n}, \theta^{\mu',n}$, this way the learner and all the actors begin with the same weights. From here on, the algorithm runs exactly the same as the DDPG until the transition (s_t, a_t, r_t, s_{t+1}) is stored to the memory R. After this takes place, if the *weights_container*^{n} is not empty, the learner weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ are taken from the container and copied to the actor weights $\theta^{Q,n}, \theta^{Q',n}, \theta^{\mu,n}, \theta^{\mu',n}$. This updates the actor weights every step in the episode, in the case the learner has sent its weights. Afterwards the *gradients_container*^{n} is checked, and if it is not full, the learning procedure defined in the DDPG begins, so a mini batch is sampled from R and the Bellman equations (Equation 3.3) are calculated. But instead of training the actor networks, only the gradients are calculated. The DDPG algorithm already explains how the actor network gradients $\nabla_{\theta^\mu} J^n$ are calculated Equation 3.5, but not the critic gradients. To get the critic gradients, the loss function Equation 3.4 has to be differentiated with respect to the critic weights $\theta^{Q,n}$, and then its sign changed to negative in order to define the optimisation as a minimisation.

$$\nabla_{\theta^Q} L^n = -\frac{1}{N} \sum_{i=1}^{N} \nabla_{\theta^{Q,n}} (y_i - Q^n(s, a|\theta^{Q,n}))^2|_{s=s_i, a=a_i} \tag{3.8}$$

After calculating both gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$, they are put in the *gradients_container*^{n} so the learner can take them. From this point the algorithm proceeds as the DDPG.

The learner:

First of all the model critic $Q^L(s, a|\theta^{Q,L})$ and actor $\mu^L(s|\theta^{\mu,L})$ learner networks are initialised, then the target critic Q'^L and actor μ'^L with a copy of the model weights $\theta^{Q',L} \leftarrow \theta^{Q,L}$ and $\theta^{\mu',L}$. Subsequently a *weights_container* and a *gradients_container* is initialised for each actor. Then all the actors N are initialised and copy of the learner weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ is put into all the *weights_container*. Afterwards the learner waits until all the *weights_container* are empty, in order to be sure that the actors have taken the weights.
Now an infinite loop begins. First an empty list called *gradients_list* is set. Then for each of the actors its *gradients_container*^{n} is checked, and if it is not empty the gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$ are taken and appended to the *gradients_list*. So the *gradients_list* will have a maximum length of the number of actors N and a minimum length of 0, in the case no actor have put gradients into its *gradients_container*^{n}. Then if the *gradients_list* is not empty, is to say there is at least one pair of gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$, all the critic gradients in the list are averaged $\overline{\nabla_{\theta^Q} L}$ and also all the actor gradients $\overline{\nabla_{\theta^\mu} J}$. Afterwards, the learner networks $Q^L(s, a|\theta^{Q,L})$ and $\mu^L(s|\theta^{\mu,L})$ are updated with this averaged gradients. Consecutively the target networks of the learner Q'^L and μ'^L are updated using the Equation 3.6 and Equation 3.7. Finally all the *weights_container* are emptied, if they are not empty, and a copy of the learner weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ is put into them. Here the infinite loop starts again.

Algorithm 3 Learner with N actors

Initialise critic $Q^L(s, a|\theta^{Q,L})$ and actor $\mu^L(s|\theta^{\mu,L})$ learner networks
Initialise target critic Q'^L and actor μ'^L with weights $\theta^{Q',L} \leftarrow \theta^{Q,L}$ and $\theta^{\mu',L} \leftarrow \theta^{\mu,L}$
Initialise $weights\_container^{1...N}$ and $grads\_container^{1...N}$
for $n = 1$ **to** N **do**
  Initialise actor n
end for
Put weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ in all $weights\_container^{1...N}$
Wait until all $weights\_container^{1...N}$ are empty
loop
  Set *gradients_list* as an empty list
  for $n = 1$ **to** N **do**
    if $grads\_container^n$ is **not** empty **then**
      Take gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$ from $grads\_container^n$
      Append taken gradients in *gradients_list*
    end if
  end for
  if *gradients_list* is **not** empty **then**
    Average critic gradients in the list: $\overline{\nabla_{\theta^Q} L}$
    Average actor gradients in the list: $\overline{\nabla_{\theta^\mu} J}$
    Update $Q^L(s, a|\theta^{Q,L})$ and $\mu^L(s|\theta^{\mu,L})$ with the averaged gradients and learning rates η_{actor}, η_{critic}
    Update learner target networks with τ:

$$\theta^{Q',L} \leftarrow \tau\theta^{Q,L} + (1-\tau)\theta^{Q',L}$$
$$\theta^{\mu',L} \leftarrow \tau\theta^{\mu,L} + (1-\tau)\theta^{\mu',L}$$

    Clear all $weights\_container^{1...N}$
    Put weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ in all $weights\_container^{1...N}$
  end if
end loop

Algorithm 4 Distributed Actor n

Initialise critic $Q^n(s, a|\theta^{Q,n})$ and actor $Q^n(s|\theta^{\mu,n})$
Initialise target critic Q'^n and actor μ'^n
Wait while *weights_container*^{n} is empty
Take learner weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ from *weights_container*^{n}
Copy learner weights $\theta^{Q,n}, \theta^{Q',n} \leftarrow \theta^{Q,L}$ and $\theta^{\mu,n}, \theta^{\mu',n} \leftarrow \theta^{\mu,L}$
Initialise replay memory R^n with capacity C
for *episode* $= 1$ **to** M **do**
  Initialise environment and get initial state s_1
  for $t = 1$ **to** T **do**
    With probability ϵ^n select a random action a_t or elsewhere $a_t = \mu^n(s_t|\theta^{\mu,n})$
    if $\epsilon^n > \epsilon^n_{min}$ **then** $\epsilon^n \leftarrow \delta\epsilon^n$
    Execute action a_t and obtain reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in memory R^n
    if *weights_container*^{n} is **not** empty **then**
      Take learner weights $(\theta^{Q,L}, \theta^{\mu,L}, \theta^{Q',L}, \theta^{\mu',L})$ from *weights_container*^{n}
      Update actor weights with learner weights:
      $$\theta^{Q,n} \leftarrow \theta^{Q,L},\ \theta^{\mu,n} \leftarrow \theta^{\mu,L},\ \theta^{Q',n} \leftarrow \theta^{Q',L},\ \theta^{\mu',n} \leftarrow \theta^{\mu',L}$$
    end if
    if *gradients_container*^{n} is **not** full **then**
      Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R^n
      Calculate Bellman equation:
      $y_i = r_i + \gamma Q'^n(s_{i+1}, \mu'^n(s_{i+1}|\theta^{\mu',n})|\theta^{Q',n})$ **if** s_{i+1} is *terminal* **then** $y_i = r_i$
      Compute critic gradients with the loss:
      $$\nabla_{\theta^Q} L^n = -\frac{1}{N}\sum_{i=1}^{N} \nabla_{\theta^{Q,n}}(y_i - Q^n(s, a|\theta^{Q,n}))^2|_{s=s_i, a=a_i}$$
      Compute actor gradients:
      $$\nabla_{\theta^\mu} J^n \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q^n(s, a|\theta^{Q,n})|_{s=s_i, a=\mu^n(s_i)} \nabla_{\theta^{\mu,n}} \mu^n(s|\theta^{\mu,n})|_{s=s_i}$$
      Put gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$ in *gradients_container*^{n}
    end if
    if s_{t+1} is a *goal_state* **then**
      break loop to finish the current episode
    end if
    Next state becomes the current state $s_t \leftarrow s_{t+1}$
  end for
end for

3.3. Distributed architecture with decentralised learning

As described at the beginning of the chapter 3, this is a distributed architecture with many actors and without a central learner, that is why it has been called "Decentralised". It is an attempt to try something new and unexplored, since all the found papers related to distributed learning use a central learner. As the previous architecture, it has been designed in order to simulate different physical machines, so here the communication is based also on memory containers. Periodically the actors send gradients, generated with mini batches of their own memory, but in this case instead of being used in a central learner, they are sent back to all the actors. So each actor receives the gradients of the other actors. In a similar way, but with less frequency, this is done with their weights. This is required in order to avoid divergences after long periods of training. As in the previous architecture, sending gradients and weights fulfils the confidentiality requirement. The Figure 3.1 shows a schematic of the architecture.

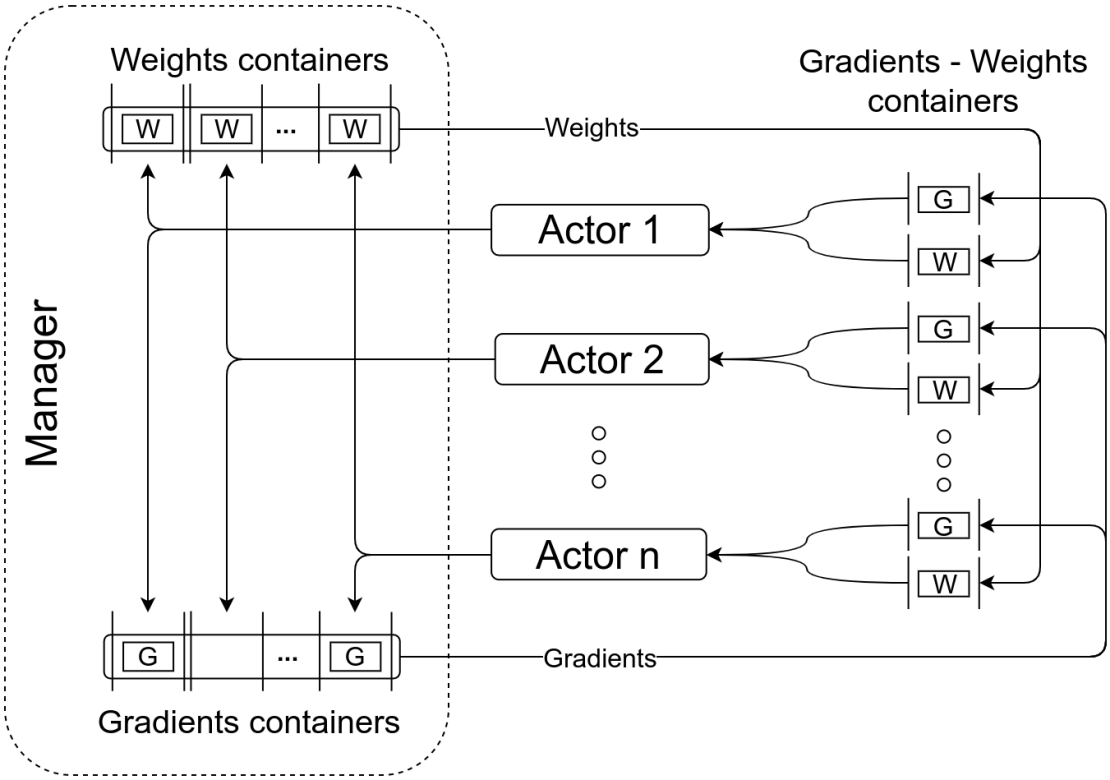


Figure 3.2.: Distributed architecture with decentralised learning.

Like previously, this architecture may be classified with the parameters Table 2.1:

- **Learning in the actor:** Actor learning
- **Number of learners:** Multiple learners
- **Acting-learning:** Coupled
- **Experience:** Actor storage
- **Actors sync:** Discrete

The architecture is divided in two parts, the actors and a manager, which ensures that the gradients and the weights are properly distributed. The algorithms written in pseudocode are described on Algorithm 5 and 6. As before the learning algorithm is the DDPG, so the training parts will be omitted unless they require clarification.

The actors:

At the beginning each actor n initialises its own critic and actor networks, model Q^n, μ^n and target Q'^n, μ'^n. In this case the manager creates the initial weights, which will be sent to all the actors to be initialised equal. So the actor waits while its $weights\_input^n$ is empty. When the $weights\_input^n$ is full, which means that the manager has sent a copy of the initial weights to the actor n, the initial weights $(\theta^Q, \theta^\mu, \theta^{Q'}, \theta^{\mu'})$ are taken from the container and copied to the actor weights $\theta^{Q,n}, \theta^{Q',n}, \theta^{\mu,n}, \theta^{\mu',n}$. From here on, the algorithm runs exactly the same as the DDPG until the transition (s_t, a_t, r_t, s_{t+1}) is stored to the memory R. After this takes place, the algorithm checks if it has received gradients, so if the $gradients\_input^n$ is not empty, the input gradients $(\overline{\nabla_{\theta^Q} L}, \overline{\nabla_{\theta^\mu} J})$ are taken from the container and the actor networks Q^n, μ^n are trained with them. Consecutively the target networks of the learner Q'^L and μ'^L are updated using the Equation 3.6 and Equation 3.7. Once this is done, if the $gradients\_output^n$ is not full the learning procedure begins. A mini batch is sampled from the memory R, the Bellman equations are calculated (Equation 3.3) and the gradients computed with Equation 3.8 and Equation 3.5. Then the gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$ are put into the $gradients\_output^n$. After each episode the $weights\_output^n$ is cleared and the actor weights $\theta^{Q,n}, \theta^{Q',n}, \theta^{\mu,n}, \theta^{\mu',n}$ are put in $weights\_output^n$, so the manager can take them. After the episode, if the $weights\_input^n$ is not empty, the averaged weights $(\overline{\theta^Q}, \overline{\theta^\mu}, \overline{\theta^{Q'}}, \overline{\theta^{\mu'}})$, calculated by the manager, are taken and copied to the actor weights $\theta^{Q,n}, \theta^{Q',n}, \theta^{\mu,n}, \theta^{\mu',n}$. Then a new episode begins.

The manager:

The main task of the manager is to receive and deliver the gradients and the weights of the actors, also it has to send the initial weights to all of them so they can be started identical. So first of all, the manager model critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ are initialised, as the target networks critic Q'^n and actor μ'^n with the same weights as the model networks. Once this is done, all the containers are created, in this case four per each actor, the $weights\_input^{1...N}$, the $weights\_output^{1...N}$, the $gradients\_input^{1...N}$ and the $gradients\_output^{1...N}$. Then all the actors are initialised. Subsequently the manager initial weights $(\theta^Q, \theta^\mu, \theta^{Q'}, \theta^{\mu'})$ are put to all the $weights\_input^{1...N}$, and the algorithm waits until all of them are empty, which means all the weights have been taken. After that the initial weights and networks can be deleted, because they will not be used any more.
Now an infinite loop begins. At the beginning an empty list called *gradients_list* is set. Then for each of the actors the $gradients\_output^n$ is check, and if it is not empty the gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$ are taken and appended to the *gradients_list*. So the *gradients_list* will have a maximum length of the number of actors N and a minimum length of 0, in the case no actor have put gradients into its $gradients\_output^n$. Then if the *gradients_list* is not empty, is to say, there is at least one pair of gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$, all the critic gradients in the list are averaged $\overline{\nabla_{\theta^Q} L}$ and also all the actor gradients $\overline{\nabla_{\theta^\mu} J}$. Afterwards, for each actor the $gradients\_input^n$ is cleared, and a copy of the averaged gradients is put into it. This way all actors have the newest gradients. Once the work with the gradients is done, the manager checks if all the $weights\_output^{1...N}$ are not empty. If they are not, then an empty list called *weights_list* is set. Then the weights $(\theta^{Q,n}, \theta^{\mu,n}, \theta^{Q',n}, \theta^{\mu',n})$ are taken from the $weights\_output^n$ for each of the actors and appended to the *weights_list*. After that, all the weights are averaged. Then all the $weights\_input^{1...N}$ are cleared and the averaged weights $(\overline{\theta^Q}, \overline{\theta^\mu}, \overline{\theta^{Q'}}, \overline{\theta^{\mu'}})$ are put in all of them. Here the infinite loop starts again.

Algorithm 5 Manager with N actors

Initialise central critic $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ networks
Initialise central target critic Q' and actor μ' with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialise $weights\_input^{1...N}$ and $weights\_output^{1...N}$
Initialise $gradients\_input^{1...N}$ and $gradients\_output^{1...N}$
for $n = 1$ **to** N **do**
  Initialise actor n
end for
Put weights $(\theta^Q, \theta^\mu, \theta^{Q'}, \theta^{\mu'})$ in all $weights\_input^{1...N}$
Wait until all $weights\_input^{1...N}$ are empty
Delete central networks Q, μ, Q' and μ'
loop
  Set *gradients_list* as an empty list
  for $n = 1$ **to** N **do**
    if $grads\_output^n$ is **not** empty **then**
      Take gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$ from $grads\_output^n$
      Append taken gradients in *gradients_list*
    end if
  end for
  if *gradients_list* is **not** empty **then**
    Average critic gradients in the list: $\overline{\nabla_{\theta^Q} L}$
    Average actor gradients in the list: $\overline{\nabla_{\theta^\mu} J}$
    Clear all $gradients\_input^{1...N}$
    Put averaged gradients $(\overline{\nabla_{\theta^Q} L}, \overline{\nabla_{\theta^\mu} J})$ in all $gradients\_input^{1...N}$
  end if
  if all $weights\_output^{1...N}$ are **not** empty **then**
    Set *weights_list* as an empty list
    for $n = 1$ **to** N **do**
      Take weights $(\theta^{Q,n}, \theta^{\mu,n}, \theta^{Q',n}, \theta^{\mu',n})$ from $weights\_output^n$
      Append taken weights in *gradients_list*
    end for
    Average weights $(\overline{\theta^Q}, \overline{\theta^\mu}, \overline{\theta^{Q'}}, \overline{\theta^{\mu'}})$
    Clear all $weights\_input^{1...N}$
    Put averaged weights $(\overline{\theta^Q}, \overline{\theta^\mu}, \overline{\theta^{Q'}}, \overline{\theta^{\mu'}})$ in all $weights\_input^{1...N}$
  end if
end loop

Algorithm 6 Decentralised actor n

Initialise critic $Q^n(s,a|\theta^{Q,n})$ and actor $\mu^n(s|\theta^{\mu,n})$ and target critic Q'^n and actor μ'^n
Wait while $weights\_input^n$ is empty
Take initial central weights $(\theta^Q, \theta^\mu, \theta^{Q'}, \theta^{\mu'})$ from $weights\_input^n$
Copy initial central weights $\theta^{Q,n}, \theta^{Q',n} \leftarrow \theta^Q$ and $\theta^{\mu',n}, \theta^{\mu,n} \leftarrow \theta^\mu$
Initialise replay memory R^n with capacity C
for $episode = 1$ **to** M **do**
  Initialise environment and get initial state s_1
  for $t = 1$ **to** T **do**
    With probability ϵ^n select a random action a_t or elsewhere $a_t = \mu^n(s_t|\theta^{\mu,n})$
    if $\epsilon^n > \epsilon^n_{min}$ **then** $\epsilon^n \leftarrow \delta\epsilon^n$
    Execute action a_t and obtain reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in memory R^n
    if $gradients\_input^n$ is **not** empty **then**
      Take gradients $(\overline{\nabla_{\theta^Q} L}, \overline{\nabla_{\theta^\mu} J})$ from $gradients\_input^n$
      Update Q^n and μ^n with the gradients and learning rates η_{actor}, η_{critic}
      Update actor target networks with τ:
        $$\theta^{Q',n} \leftarrow \tau\theta^{Q,n} + (1-\tau)\theta^{Q',n} \text{ and } \theta^{\mu',n} \leftarrow \tau\theta^{\mu,n} + (1-\tau)\theta^{\mu',n}$$
    end if
    if $gradients\_output^n$ is **not** full **then**
      Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R^n
      Calculate Bellman equation:
      $y_i = r_i + \gamma Q'^n(s_{i+1}, \mu'^n(s_{i+1}|\theta^{\mu',n})|\theta^{Q',n})$ **if** s_{i+1} is *terminal* **then** $y_i = r_i$
      Compute critic gradients with the loss:
        $$\nabla_{\theta^Q} L^n = -\frac{1}{N}\sum_{i=1}^{N} \nabla_{\theta^{Q,n}} (y_i - Q^n(s,a|\theta^{Q,n}))^2|_{s=s_i, a=a_i}$$
      Compute actor gradients:
        $$\nabla_{\theta^\mu} J^n \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q^n(s,a|\theta^{Q,n})|_{s=s_i, a=\mu^n(s_i)} \nabla_{\theta^{\mu,n}} \mu^n(s|\theta^{\mu,n})|_{s=s_i}$$
      Put gradients $(\nabla_{\theta^Q} L^n, \nabla_{\theta^\mu} J^n)$ in $gradients\_output^n$
    end if
    if s_{t+1} is a *goal_state* **then break** loop to finish the current episode
    Next state becomes the current state $s_t \leftarrow s_{t+1}$
  end for
  Clear $weights\_output^n$
  Put actor weights $(\theta^{Q,n}, \theta^{\mu,n}, \theta^{Q',n}, \theta^{\mu',n})$ in $weights\_output^n$
  if $weights\_input^n$ is **not** empty **then**
    Take averaged weights $(\overline{\theta^Q}, \overline{\theta^\mu}, \overline{\theta^{Q'}}, \overline{\theta^{\mu'}})$ from $weights\_input^n$
    Copy averaged weights: $\theta^{Q,n} \leftarrow \overline{\theta^Q}$, $\theta^{Q',n} \leftarrow \overline{\theta^{Q'}}$, $\theta^{\mu,n} \leftarrow \overline{\theta^\mu}$ and $\theta^{\mu',n} \leftarrow \overline{\theta^{\mu'}}$
  end if
end for

3.4. Distributed architecture with shared memory space

The third system developed, is a distributed architecture with many actors and one unique central learner, as the first one. The difference is, that there is no need for communication since here the actors and the learner share the same memory space, so in principle the actors have direct access to the learner networks. In this case the actors run episodes and every time they have to act they access the learner to get the action. The actors have their own memory and each of them gathers its own experiences. Every step of the episode the actor access the learner to train it with a mini batch sampled from the actor memory. In order to avoid reading and writing the learner at the same time, a lock system has been implemented which controls the learner accesses. The Figure 3.1 shows a schematic.

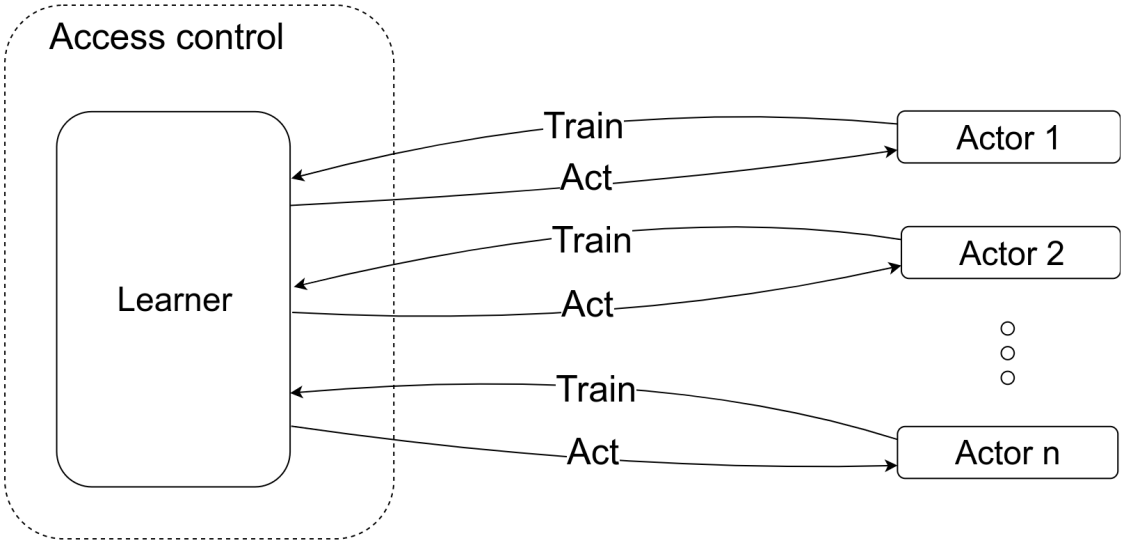


Figure 3.3.: Distributed architecture with shared memory space.

It may be classified with the parameters Table 2.1:

- **Learning in the actor:** Non-actor learning
- **Number of learners:** One learner
- **Acting-learning:** Coupled
- **Experience:** Actor storage
- **Actors sync:** Continuous

As seen, this architecture is basically the same as the one described in the A3C [8]. The main difference is that here a DDGP algorithm has been used, like in the paper "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation" [14]. A part from that, there are other minor differences, like the training frequencies, but mainly is the same. So the aim of this architecture is to use it to compare its performance with the other ones which must send information, in order to see the impact of the communication effect .

Like the first one, this architecture is divided in two parts, the learner and the actors. The algorithms written in pseudocode are described on Algorithm 7 and 8. The learning algorithm is also the DDPG, so the training parts will be omitted.

The learner:

It is used basically to initialise the shared networks, create the lock access system and start the actors. So first of all the model critic $Q^L(s, a|\theta^{Q,L})$ and actor $\mu^L(s|\theta^{\mu,L})$ networks are initialised, then the target critic Q'^L and actor μ'^L with a copy of the model weights $\theta^{Q',L} \leftarrow \theta^{Q,L}$ and $\theta^{\mu',L}$. Then the *lock* system is initialised. Finally, the actors are started sharing with them the *lock* system.

The actors:

The algorithm is exactly the same as the DDPG. The only difference is that a *lock* access is required every time an action is selected and every time the networks are updated. This way, reading/writing conflicts are avoided if two actors or more try to read/write the learner networks.

Algorithm 7 Learner with N actors and shared memory space

Initialise critic $Q^L(s, a|\theta^{Q,L})$ and actor $\mu^L(s|\theta^{\mu,L})$ learner networks
Initialise target critic Q'^L and actor μ'^L with weights $\theta^{Q',L} \leftarrow \theta^{Q,L}$ and $\theta^{\mu',L} \leftarrow \theta^{\mu,L}$
Create a *Lock* system to control the access to the learner
for $n = 1$ **to** N **do**
  Initialise actor n
end for

Algorithm 8 Distributed actor n with shared memory space

Get global *Lock* to control access to the learner
Initialise replay memory R^n with capacity C
for $episode = 1$ **to** M **do**
  Initialise environment and get initial state s_1
  for $t = 1$ **to** T **do**
    Acquire: *Lock*
      With probability ϵ^n select a random action a_t or elsewhere $a_t = \mu^L(s_t|\theta^{\mu,L})$
    Release: *Lock*
    Execute action a_t and obtain reward r_t and next state s_{t+1}
    if $\epsilon^n > \epsilon^n_{min}$ **then** $\epsilon^n \leftarrow \delta\epsilon^n$
    Store transition (s_t, a_t, r_t, s_{t+1}) in memory R^n
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R^n
    Acquire: *Lock*
      Calculate Bellman equation:
      $y_i = r_i + \gamma Q'^L(s_{i+1}, \mu'^L(s_{i+1}|\theta^{\mu',L})|\theta^{Q',L})$ **if** s_{i+1} is *terminal* **then** $y_i = r_i$
      Update learner critic by minimizing the loss with learning rate η_{critic}:

$$L^L = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q^L(s_i, a_i|\theta^{Q,L}))^2$$

      Update learner actor with policy gradient and learning rate η_{actor}:

$$\nabla_{\theta^\mu} J^L \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_a Q^L(s, a|\theta^{Q,L})|_{s=s_i, a=\mu^L(s_i)} \nabla_{\theta^\mu}\mu^L(s|\theta^{\mu,L})|_{s=s_i}$$

      Update learner target networks with τ:

$$\theta^{Q',L} \leftarrow \tau\theta^{Q,L} + (1-\tau)\theta^{Q',L}$$
$$\theta^{\mu',L} \leftarrow \tau\theta^{\mu,L} + (1-\tau)\theta^{\mu',L}$$

    Release: *Lock*
    if s_{t+1} is a *goal_state* **then**
      break loop to finish the current episode
    end if
    Next state becomes the current state $s_t \leftarrow s_{t+1}$
  end for
end for

4. Implementation

This chapter explains how the algorithms, described previously, have been implemented to be evaluated.

4.1. Infrastructure

The whole system has been built under the Ubuntu 16.04 LTS[1] operating system using Python 3.5.2[2] as the programming language. Ubuntu has been chosen because of its flexibility and easiness to work with Python among many other advantages. Python has been selected because the robotics simulator used, explained in the next section, requires Python to work. In addition, it is one of the languages most used nowadays and it has a lot of libraries, which complement it and allow faster and better results.

In order to manage all the required neural networks, the open source machine learning platform TensorFlow[3] has been used. In some parts of the code, Keras[4] the high-level neural networks API has been used to define the networks structure and their training. Both libraries, TensorFlow and Keras work with Python.

Finally, a computer from the *Technische Universität München* located in the *Chair of Robotics, Artificial Intelligence and Real-time Systems* has been used to run all the tests and experiments. This computer has been accessed via a Secure Shell (SSH) protocol.

4.2. Simulation environment

The simulation environment used in this project is the Robotics platform[5] provided by Gym from OpenAI[3]. Gym is a toolkit for developing and comparing RL algorithms and its robotics platform provides a robotics simulator with different possible tasks. The robotics platform is built under the physics simulator MuJoCo[6], which is required to make the platform work.

[1] https://ubuntu.com/ [2] https://www.python.org/ [3] https://www.tensorflow.org/
[4] https://keras.io/ [5] http://gym.openai.com/envs/#robotics [6] http://www.mujoco.org/

The robotics platform offers eight different environments, four of them use a robotic arm to perform the tasks and the other four use a robotic hand. The Figure 4.1 shows the eight possible environments.

Figure 4.1.: OpenAI gym robotics environments

This project uses only the robotic arm environments, and not all of them, but only the "FetchReach-v1" and the "FetchPickAndPlace-v1". The task in "FetchReach-v1" consists in moving the robot TCP(Tool center point), in this case the centre of the gripper, to a random target position. Otherwise, the task in "FetchPickAndPlace-v1" consists in grasping a cube, which has been placed randomly on the table, and moving it to a random target position. The Figure 4.2 shows a schematic of the tasks. The first one, considered a trivial task, will be used just to check if an algorithm works, the second one is considered a complex task and will be used to actually asses the performance of the algorithms.

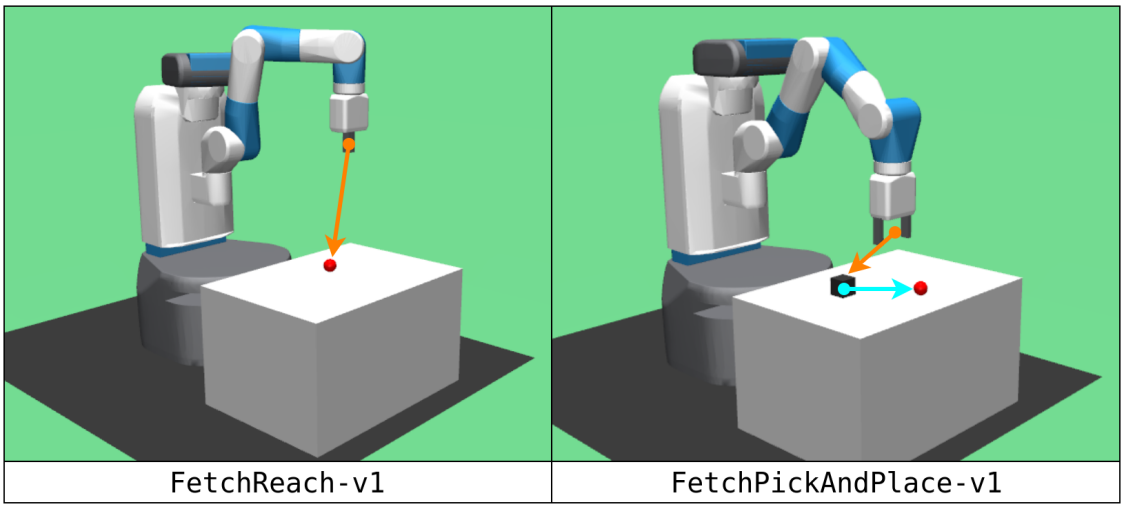


Figure 4.2.: Tasks description

The interaction with the environments is very simple. After being initialised, the platform works as a discrete sequential system, in each step an action is required as an input, then the effect of the action is simulated and the platform outputs the new state data. The Algorithm 9 shows the interaction structure with the platform.

Algorithm 9 Interaction with the robotics platform

```
Initialise the Robotics Platform
Get the initial output: Observation, Reward, Done, Information
for each step in Steps do
    Input an action to the Robotics Platform
    Robotics Platform outputs: Observation, Reward, Done, Info
end for
```

The input **Action** is given as an array with the physical data of the action. The output data is obtained divided in four components:

- **Observation:** Array with the physical data of the new state
- **Reward:** Numerical reward of the transition
- **Done:** Boolean value which tells if the episode is done, is to say, if the target has been reached or if the maximum allowed number of steps has been exceeded.
- **Info:** Dictionary with additional information

Depending on the environment, the structure of the input and output data may differ. For example, the environment "FetchReach-v1" does not have any cube, so the output will not contain any data about it, therefore the observation array will have a different length.

In order to clarify the notation used to describe the physical data of the input and the output, and to show where the TCP is, the Figure 4.3 pictures this information on a robot image.

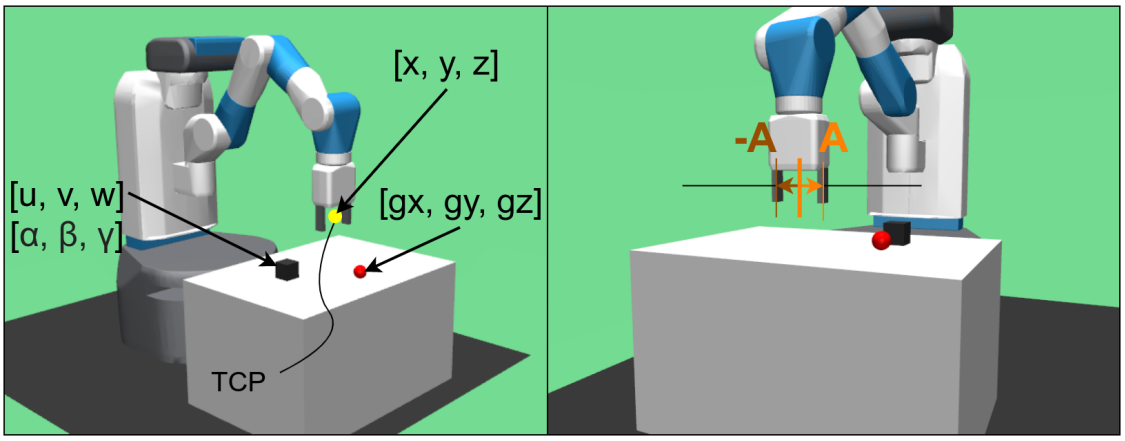


Figure 4.3.: Axes naming

The input action, in both environments, is expected as an increment. The first three values define an increment of the current position of the TCP and the last value defines an increment of the gripper opening. The Table 4.1 shows the arrays for both environments.

	FetchReach-v1	**FetchPickAndPlace-v1**
TCP and Gripper position increment	[$\triangle x$, $\triangle y$, $\triangle z$, -]	[$\triangle x$, $\triangle y$, $\triangle z$, $\triangle A$]

Table 4.1.: Expected input action array

On the other hand, the output data of the observation is very different from one task to the other, since one of the tasks uses a cube and the other does not. The Table 4.2 shows the observation array obtained in both environments.

	FetchReach-v1	**FetchPickAndPlace-v1**
Goal position	$[gx, gy, gz]$	$[gx, gy, gz]$
Achieved goal	$[x, y, z]$	$[u, v, w]$
TCP position	$[x, y, z]$	$[x, y, z]$
Object position	-	$[u, v, w]$
Object relative position	-	$[u - x, v - y, w - z]$
Gripper state	$[A, A]$	$[A, A]$
Object rotation	-	$[\alpha, \beta, \gamma]$
Object velocity	-	$[\dot{u}, \dot{v}, \dot{w}]$
Object relative velocity	-	$[u \dot{-} x, v \dot{-} y, w \dot{-} z]$
TCP velocity	$[\dot{x}, \dot{y}, \dot{z}]$	$[\dot{x}, \dot{y}, \dot{z}]$
Gripper velocity	$[-\dot{A}, \dot{A}]$	$[-\dot{A}, \dot{A}]$

Table 4.2.: Expected content of the output observation array

4.3. Networks

As explained previously, the libraries used for the neural networks are Keras and Tensorflow. Keras has been used mainly for the high level operations, such as the building of the networks structure and the networks training with target values. On the other hand, TensorFlow has been required for lower level operations, such as gradients calculations and training, which has been used in the distributed algorithms.

Since all the learning algorithms implemented in this work are based on the DDPG algorithm, all of them require at least two networks to work. The critic network Q and the actor network μ. Also there are the target networks, but they are a copy of the model networks, so they are not considered different. All the algorithms developed in this work use the exact same structure for their critic and actor networks. The only difference depends on the task performed, because the action and state sizes differ, and they are the inputs and outputs of the networks.

4.3.1. Actor network μ

This network has the state as input. For the task "FetchReach-v1" the state is an array of nine elements, which contains the goal position, the TCP position and the TCP velocity. The gripper state and velocity are not required, because the gripper is not used in this task. The achieved goal is not required either, because its the same vector as the TCP position. For the task "FetchPickAndPlace-v1" the state is an array of twenty-eight elements, which contains the goal position, the TCP position, the object position, the object relative position, the gripper state, the object rotation, the object velocity, the object relative velocity, the TCP velocity and the gripper velocity. As in the previous task, the achieved goal is not required, because it is the same vector as the object position. (See Table 4.2)

The output of this network is the action. For the task "FetchReach-v1" the action has only three elements, the vector with the TCP position increments, since the gripper is not used. For the task "FetchPickAndPlace-v1" the action has four elements, the vector with the TCP position increments plus the position increment of the gripper. (See Table 4.1)

Independently from the input and output size, the other parts of the network are identical across all the algorithms. This network consist of three hidden fully connected layers of five-hundred neurons each one and with a *ReLU* activation function. The output layer, with the size of the actions, has a *tanh* activation function in order to

constrain the output $actions \in [-1, 1] \subset \mathbb{R}$. The Figure 4.4 shows a schematic of the actor network μ structure.

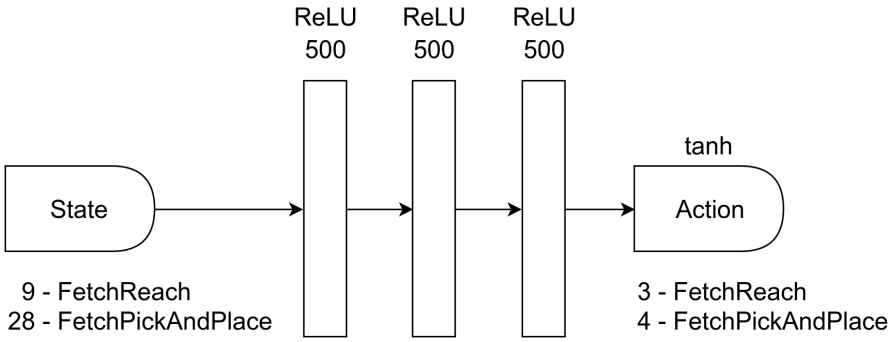


Figure 4.4.: Actor network μ

4.3.2. Critic network Q

This network has the state and the action as input, which have already been described for the actor network μ. Each of these two inputs is fully connected to a layer of five-hundred neurons with a *ReLU* activation function. The output of these two layers is concatenated and fully connected to a layer of five-hundred neurons and also with a *ReLU* activation function. This last layer is fully connected again to a layer of five-hundred neurons with a *ReLU* activation function. Finally this layer is fully connected to one neuron, the output, which is the Q-value. The activation function of this last output layer is *linear*, since the Q-value can be any real number $\mathbb{R}$. The Figure 4.5 shows a schematic of the critic network Q structure.

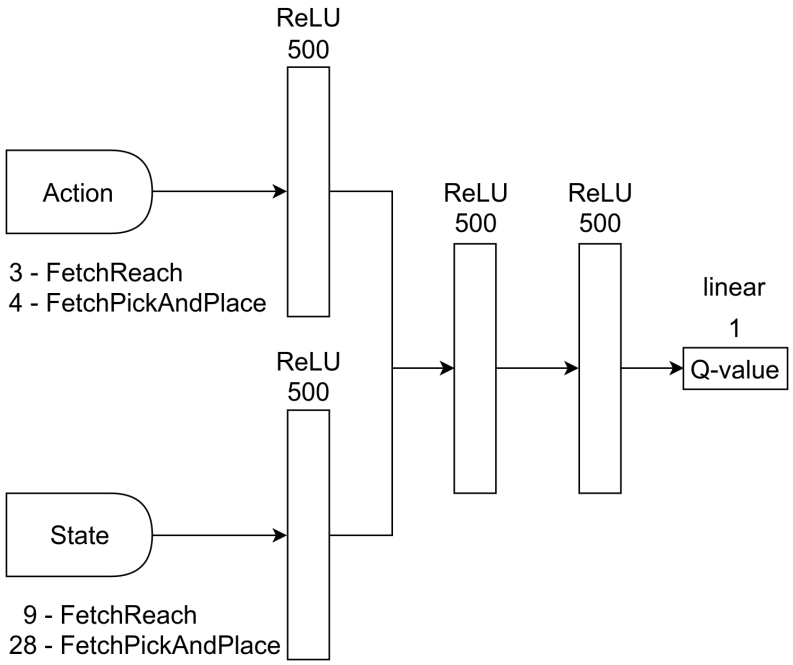


Figure 4.5.: Critic network Q

As it has been seen in the previous chapters, the methods used to train the two networks are different. To train the actor μ, its weights are modified in order to maximise the Q-value using the gradients from the Equation 3.5. On the other hand, to train the critic network Q the minimisation of a loss function is used. The loss function defined in the algorithms is the *mean squared error*, Equation 3.4. But in the real implementation a different loss function has been used, the *Logcosh*, described in the Equation 4.1.

$$L = \frac{1}{N}\sum_{i=1}^{N}\log(\cosh(y_i - Q(s_i, a_i|\theta^Q))) \tag{4.1}$$

This loss function is similar to the *mean squared error*, but instead of having an exponential shape for all the possible error values, it has an exponential shape for small errors around zero and a linear shape for large errors. The *Logcosh* loss function shows better performance in the training of the networks, thus it has been chosen instead the classical *mean squared error*.

Finally, *Adam* has been the optimiser used in all the networks, which is an enhanced version of the classical *SGD* (Stochastic gradient descent).

4.4. Tools

To assist the different parts of the program different tools have been implemented. Some of them encapsulated in classes and some others developed as basic functions. The Figure 4.6 shows their main structure.

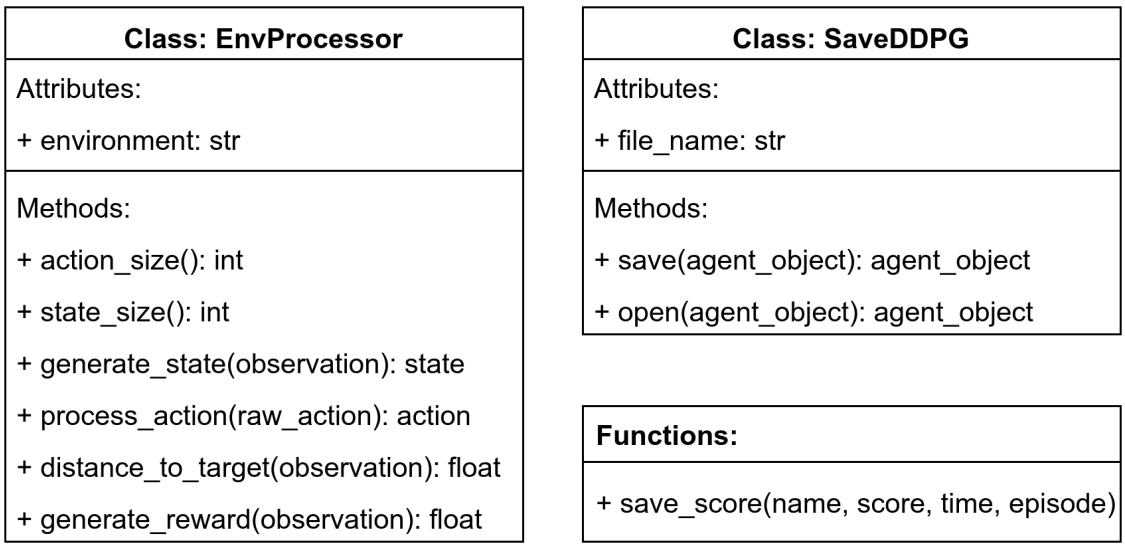


Figure 4.6.: Tools

4.4.1. Save score

This tool is a basic function, which will be used to save the information of the learning process. This function opens or creates a file with the given name and adds a row of data at the end of it. Each row of data will be written with the other given elements: score, time and episode.

4.4.2. SaveDDPG

This is a class used to save the weights of the neural networks, in order to be able to use them after the training. The class must be initialised with the required parameter, which is the file name. It has only two methods, save and open. This methods require an agent object, which is an instance of the class that contains the neural networks. When the save method is called, the weights are extracted from the given object to be saved. The other way round, when the open method is called, the weights are loaded into the given object.

4.4.3. EnvProcessor:

This tool, implemented as a class, works as an interface between the RL, is to say the intelligence of the program, and the simulation environment. This is required, because the output generated by the simulation environment (See Table 4.2) must be conditioned before being used by the RL. The other way round, the actions generated by the RL must be adapted to fit into the simulator input (See Table 4.1).

This class is initialised with a required parameter, which is a string with the environment name, for example "FetchReach-v1". This is necessary, because depending on the environment used, the methods will work differently.

Methods:

- **action_size():** Returns the action size of the given environment when the class was initialised.
- **state_size():** Return the state size of the given environment when the class was initialised.
- **generate_state(observation):** Given the observation generated by the simulator, returns an array with the size of the state, ready to be input in the RL networks.
- **process_action(raw_action):** Given the raw action generated by the RL networks, returns this action processed to fit as an input in the simulator.
- **distance_to_target(observation):** Given the observation generated by the simulator, returns the distance from the TCP or the cube to the target position.
- **generate_reward(observation):** Instead of using the reward generated by the simulator, in this project a custom reward will be used. This method returns the custom reward given the observation generated by the simulator.

In addition to this tools, which are used directly in the training process, another tool has been developed to plot the results obtained. This tool plots the results saved with the tool "Saved score". Basically it consist in a plotter with the Y axis showing the percentage of success and the X axis can be the number of episodes or the time elapsed. Also the results can be averaged over the measurements to enable smother curves in order to easy the analysis. The Figure 4.7 shows an example of a plot.

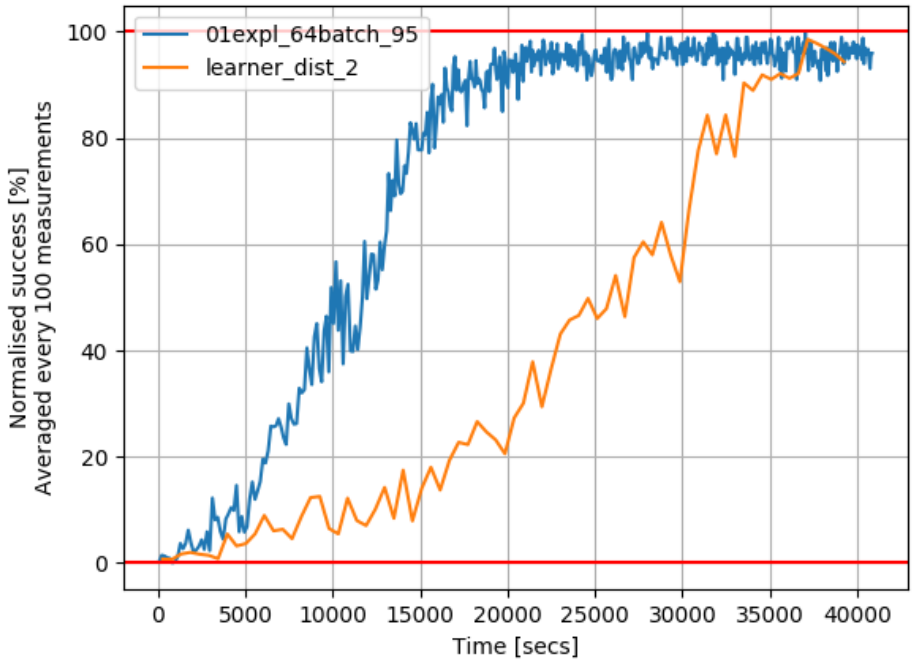


Figure 4.7.: Plot example

4.5. Rewards

As said previously, the rewards given by the simulator are not used in this project, instead custom ones have been used. The rewards used in this work are defined depending on the distance between the target position and the position of the TCP (in the "FetchReach-v1" task) and the position of the cube (in the "FetchPickAndPlace-v1" task). As mentioned before, this rewards are calculated using the tool "EnvProcessor". The Equation 4.2 shows how the reward is calculated for the task "FetchReach-v1" and the Equation 4.3 for the task "FetchPickAndPlace-v1". (See Table 4.2)

$$r = -\sqrt{(x - gx)^2 + (y - gy)^2 + (z - gz)^2} \tag{4.2}$$

$$r = -\left[\sqrt{(x - u)^2 + (y - v)^2 + (z - w)^2} + \sqrt{(u - gx)^2 + (v - gy)^2 + (w - gz)^2}\right] \tag{4.3}$$

4.6. Score

To quantify how good the tasks are performed a numerical score have been defined. It is basically the distance between the TCP or the cube and the goal position at the end of the episode, compared with the distance at the beginning of the episode and expressed in percentage. Since the goal position is a vector with real values $\mathbb{R}$ and the TCP or cube position vector too, they can not be directly compared with a pure equality and a margin is required, to consider that the goal has been reached. The simulator defines this value as 0.05m by default, and this is the one that will be used in this work. In some cases the final position of the TCP or the cube can be closer than 0.05m or further than the initial position, therefore some scores could be higher than one-hundred percent or lower than zero percent, to solve this, the scores are clipped between zero and one-hundred. The Equation 4.4 shows the score calculation for the task "FetchReach-v1" and the Equation 4.5 shows it for the task "FetchPickAndPlace-v1". (See Table 4.2)

$$s = 100 \left[\frac{\sqrt{(x_i - gx)^2 + (y_i - gy)^2 + (z_i - gz)^2} - \sqrt{(x_f - gx)^2 + (y_f - gy)^2 + (z_f - gz)^2}}{\sqrt{(x_i - gx)^2 + (y_i - gy)^2 + (z_i - gz)^2} - 0.05} \right] \tag{4.4}$$

$$s = 100 \left[\frac{\sqrt{(u_i - gx)^2 + (v_i - gy)^2 + (w_i - gz)^2} - \sqrt{(u_f - gx)^2 + (v_f - gy)^2 + (w_f - gz)^2}}{\sqrt{(u_i - gx)^2 + (v_i - gy)^2 + (w_i - gz)^2} - 0.05} \right] \tag{4.5}$$

4.7. Testing

In order to test the algorithms a testing function has been created. This function is called during the training by the algorithms periodically in order to know the algorithm performance. It can also be called externally to test agents already trained. The Figure 4.8 shows the arguments required and the return of the function.

Test function:
+ test(agent, environment, episodes): episodes averaged score

Figure 4.8.: Test function

The function takes the given agent and tests it in the given environment for the given number of episodes. This is done with a zero exploration rate $\epsilon = 0$, without saving anything in the replay memory and without training the agent. For each of the episodes

the function calculates the score, using the Equation 4.4 or the Equation 4.5, and after all the episodes the scores are averaged. Then the function returns the averaged scores. Of course, if the given number of episodes is one, the return will be the score of this only episode.

4.8. Agents

All the RL learning parts of the algorithms have been implemented in classes, which will be called agents. Each of this classes has the attributes and methods required for the training, interaction with the simulation and interaction with the developed tools.

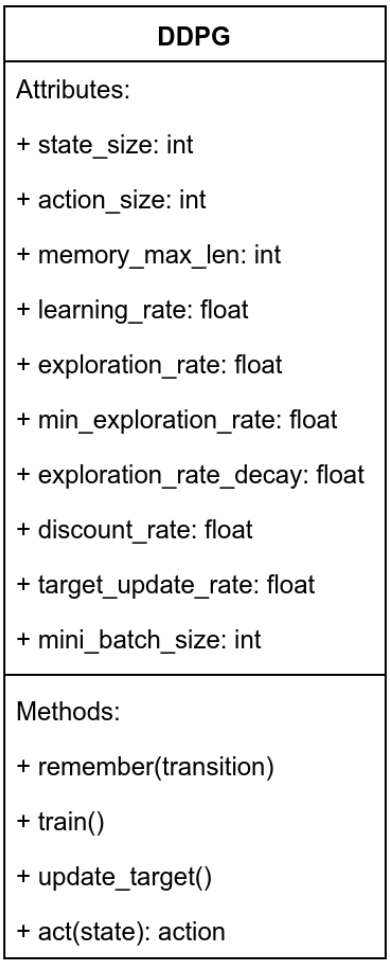


Figure 4.9.: DDPG

4.8.1. Deep Deterministic Policy Gradient

The Figure 4.9 schematises the DDPG agent. The attributes "state_size", "action_size" and "learning_rate" are used to create the networks when the class is initialised. The "state_size" and "action_size" will define the input and output sizes of the network. As it can be seen, all the other attributes correspond to the hyper parameters defined in the section 3.1. The only difference is that, in this work, the same "learning_rate" has been used for the two networks.

The method "remember(transition)" stores the given transition into the replay memory of the class. The method "train()" picks a random sample of transitions, with the size of the mini batch, from the replay memory. Then it trains the critic and actor networks with that sample. The method "update_target()" updates the target networks. Finally, the method "act(state):action" generates an action given an state and updates the value of the "exploration_rate". The action can be randomly generated, with a probability defined by the "exploration_rate", or if not it is generated by the actor network.

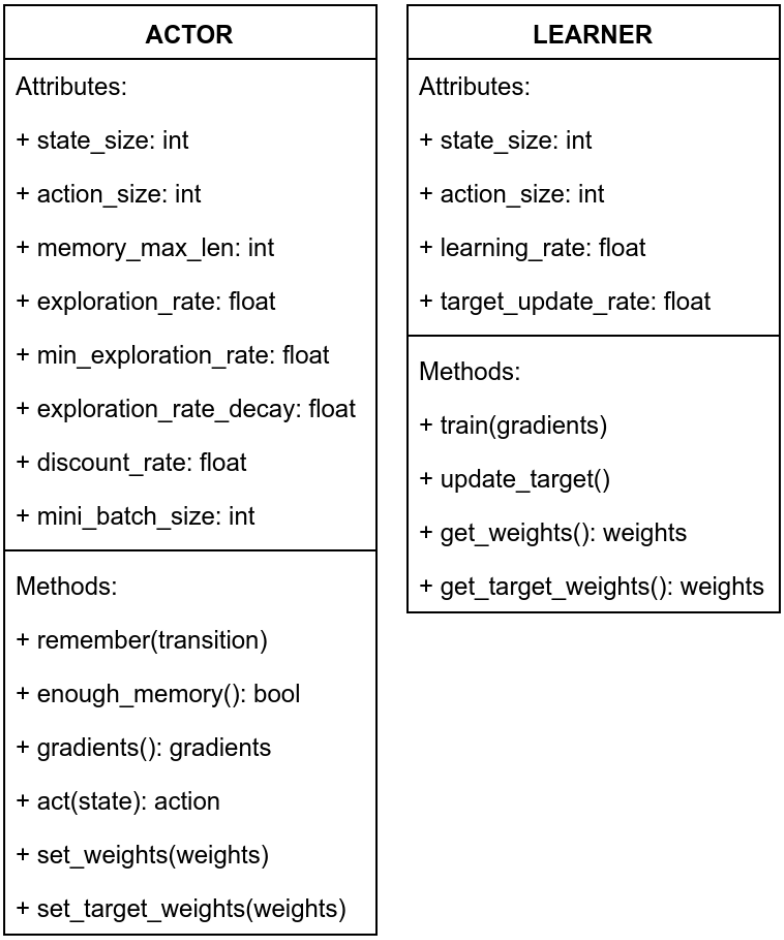


Figure 4.10.: Centralised learning

4.8.2. Distributed architecture with centralised learning

In this architecture two agents are required, one for the central learning called "LEARNER" and one for the multiple actors called "ACTOR".(Figure 4.10)

The "LEARNER" contains all the attributes required just for the training of the networks. Thus, it has the "state_size", "action_size", "learning_rate" and "target_update_rate". It has the method "train(gradients)" which trains the networks with the given gradients received from the actors. Like the previous agent, it has the method "update_target()"

which updates the target networks. In order to obtain the weights to send them to the actors, it has the methods "get_weights():weights" and "get_target_weights():weights", which return the weights of the model networks and the target networks.

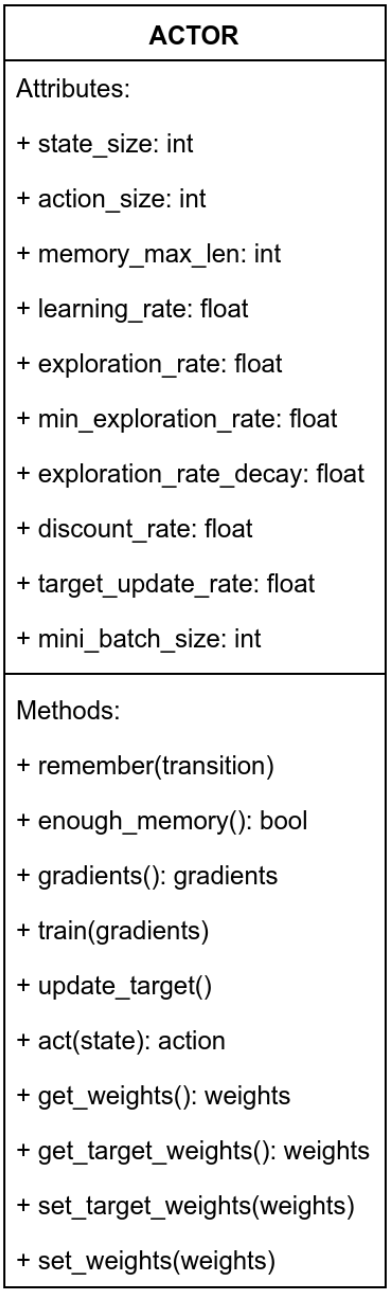


Figure 4.11.: Decentralised learning

The "ACTOR" contains a copy of the learner networks, therefore it requires the attributes "state_size" and "action_size" to initialise it. A part from these two attributes, it has all the attributes related to the replay memory, the acting and the RL calculations to generate the gradients. The method "remember(transition)" saves the given transition to the replay memory. The method "enough_memory():bool" returns *true* if the length of the replay memory is equal or larger than the mini batch size. This is used to indicate if there are enough samples in the memory so the calculation of the gradients may begin. The method "gradients():gradients" takes a sample from the memory replay and returns the gradients calculated with it, so they can be send to the learner. The "act(state):action" returns an action given a state, taking care of the "exploration_rate" as before. Finally, the methods "set_weights(weights)" and "set_target_weights(weights)" copy the given weights to the model and target networks.

4.8.3. Distributed architecture with decentralised learning

In this case there is only one class, called "ACTOR" that contains everything.(Figure 4.11) This class contains exactly the same attributes and methods that the "LEARNER" and "ACTOR", of the distributed architecture with centralised learning, would have, if they were merged. This is because, in this architecture, all the actors are learners at the same time.

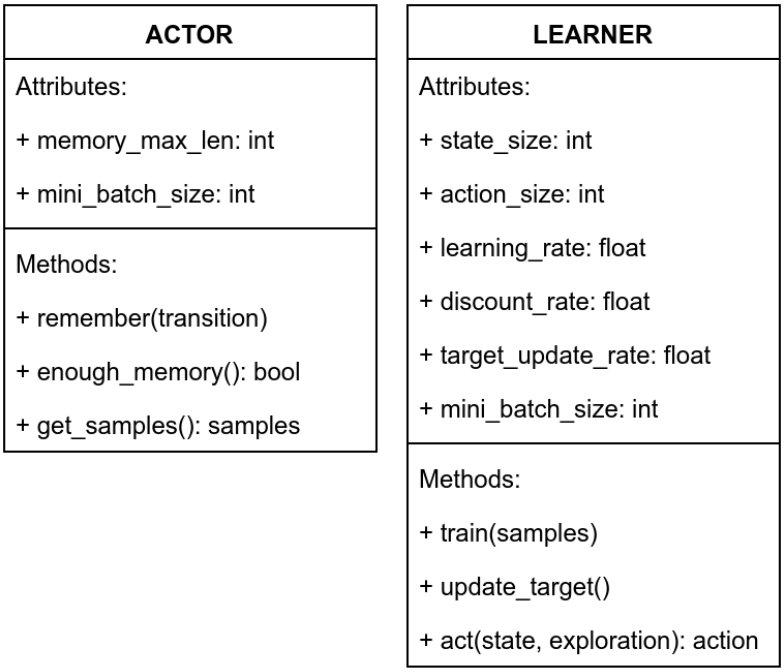


Figure 4.12.: Shared memory learning

4.8.4. Distributed architecture with shared memory space

As in the distributed architecture with centralised learning, here two agents are required, called "LEARNER" and "ACTOR".(Figure 4.12)

The "LEARNER" has the attributes "state_size" and "action_size" to create the networks when it is initialised. All the other attributes are related to the training of the networks. The last attribute "mini_batch_size" is required to know the expected size of the samples that will receive by the actors. It has the "train(samples)" method, which will train the networks with the given samples. As the previous learners, it has the method "update_target()" in order to update the target networks. Finally, the "act(state, exploration):action" method is used to generate an action given a state. In this case it has another parameter, called "exploration", this is because, in this algorithm the acting belongs to the learner instead to the actor, but each of the actors has a different exploration rate, so the exploration rate can not be updated by the learner. This parameter is updated externally in each of the actors and given as a parameter when acting.

The "ACTOR", in this case, is basically a class used to manage the replay memory. It has only two attributes, the "memory_max_len" used to create the memory when the class is initialised, and the "mini_batch_size" used to know how many samples must be picked from the memory when required. It has only three methods, "remember(transition)" to save a transition in the memory, "enough_memory():bool" to know if the memory is larger than the mini batch and the method "get_samples():samples" to get samples for the training.

4.9. Program construction

Now that all the elements required for the implementation have been described, the final structure of the programs may be explained.

In all the architectures, the main program is called "Train". In the DDPG algorithm, the train program runs directly all the loops that take part in the learning. In the other architectures, the distributed ones, the train program spawns the threads or processes for the parallelising of the learning.

The architectures that simulate different physical machines, is to say the distributed with centralised learning and the distributed with decentralised learning, use the built-in library *multiprocessing* from python to run the programs in parallel. This library does not use shared memory space, so the information must be sent from on process to the other. To send the information among processes, is to say the weights and the gradients, the *queue* objects have been used. These queues defined with a maximum length of one element work as the containers defined in the algorithms description.

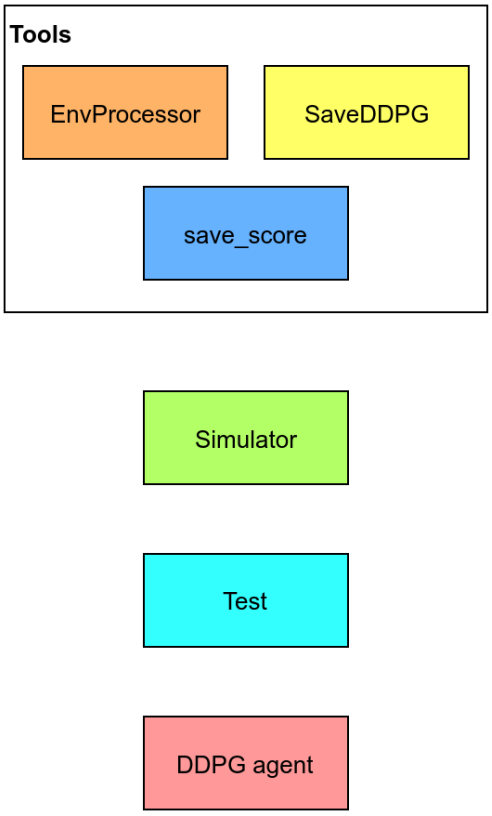
On the other hand, the architecture that simulates a shared memory space, is to say the distributed architecture with shared memory space, uses the built-in library *threading* from python to run the programs in parallel. This library allows shared memory, so all the actors can access the learner. In addition, the lock control system used is the one provided by this library too. This library has a problem, it does not run the different processes in different processors as the multiprocessing library does, therefore this could have an effect in the results since it is not that efficient. In the end, this library has been chosen anyway because of the complexity of doing it with any other one. Actually there is a way to share memory space using the library multiprocessing, but the data types allowed are too low level, so it would be very complex to share high level objects like learner instances.

As mentioned before, a part from the built-in python libraries, other libraries have been used. *Keras* and *TensorFlow* to build all the parts related to the training of the networks. *Gym* running on top of *mujoco-py* as the robot simulator. And finally *matplotlib* to plot all the data.

The Figure 4.13, Figure 4.14, Figure 4.15 and Figure 4.16 show accurately the program structure for each of the four architectures. They are written in pseudo code using the previously defined tools and agents. A color code is used in order to facilitate the elements identification.

Note: Some ideas to write the code were taken from the code repositories *deep-q-learning*[5], *keras-ddpg*[12] and *A3C-Tensorflow*[6] stored in GitHub[7].

[7] https://github.com/



Train DDPG

Initialise Simulator
Initialise EnvProcessor
Initialise SaveDDPG
Instatiate DDPG agent

Loop Episodes
observation = Reset Simulator
state = EnvProcessor.generate_state(observation)
Loop Steps
raw_action = DDPG.act(state)
action = EnvProcessor.process_action(raw_action)
observation = Simulator(action)
next_state = EnvProccessor.generate_state(observation)
reward = EnvProcessor.generate_reward(observation)
state = next_state
target_distance = EnvProcessor.distance_to_target(observation)
if target_distance < 0.05 **then** terminal
DDPG.remember(state, raw_action, reward, next_state, terminal)
DDPG.train()
DDPG.update_target()
if terminal **then break loop**
End Loop Steps
Every n seconds SaveDDPG.save(DDPG)
Every m seconds:
score = Test(DDPG, Simulator, 1 episode)
save_score(score)
Print information
End Loop Episodes

Figure 4.13.: DDPG

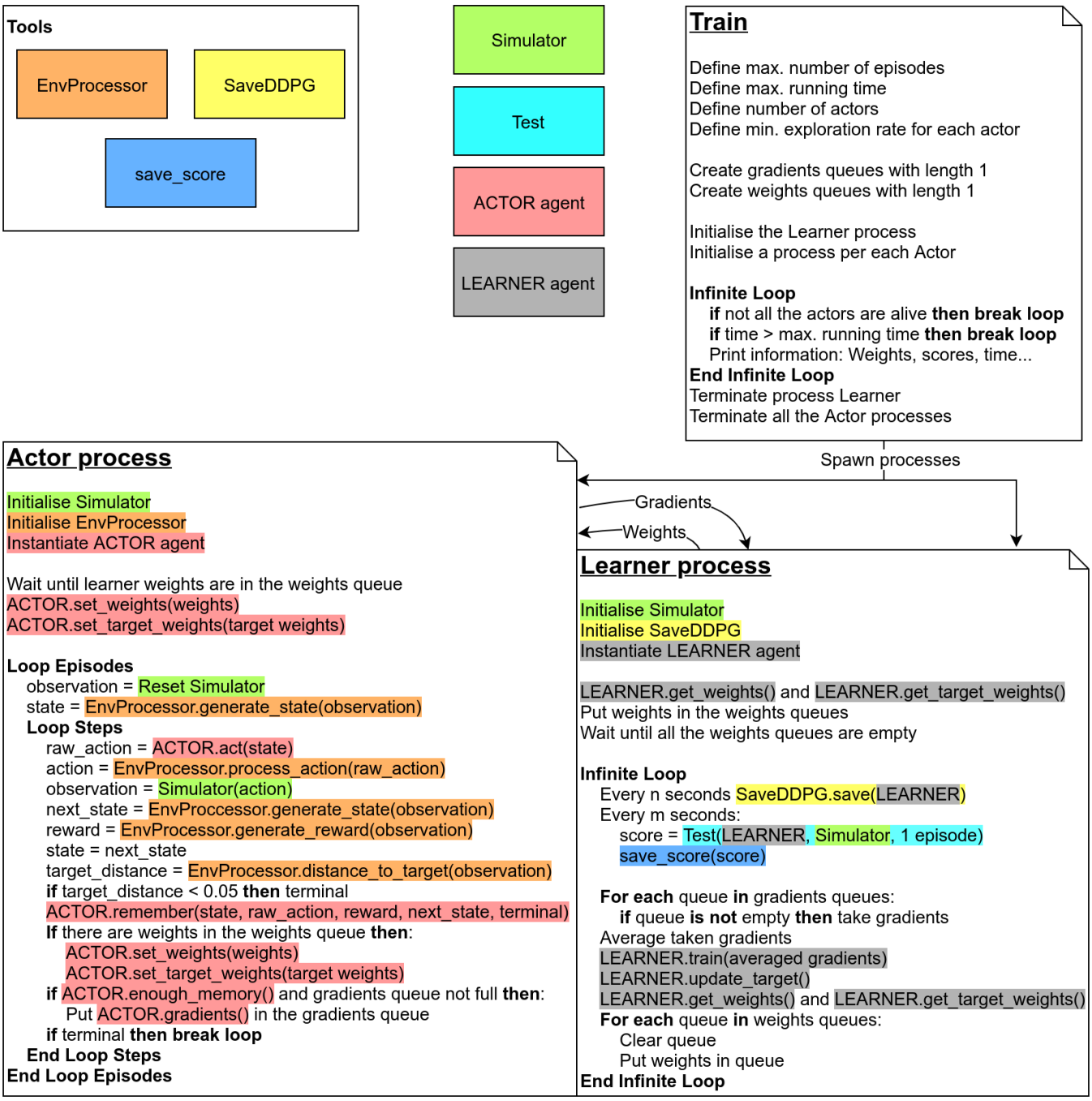


Figure 4.14.: Centralised learning

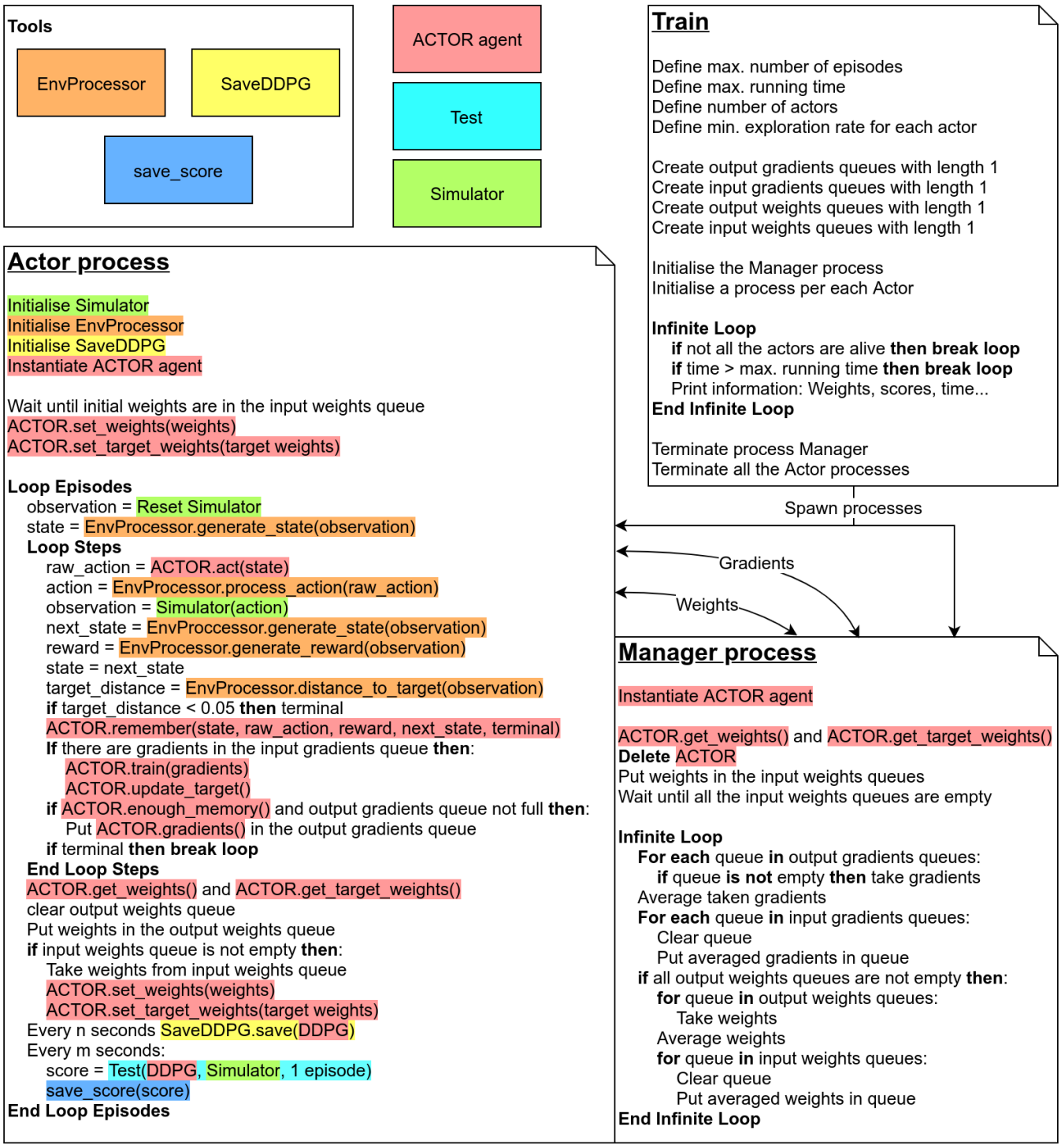


Figure 4.15.: Decentralised learning

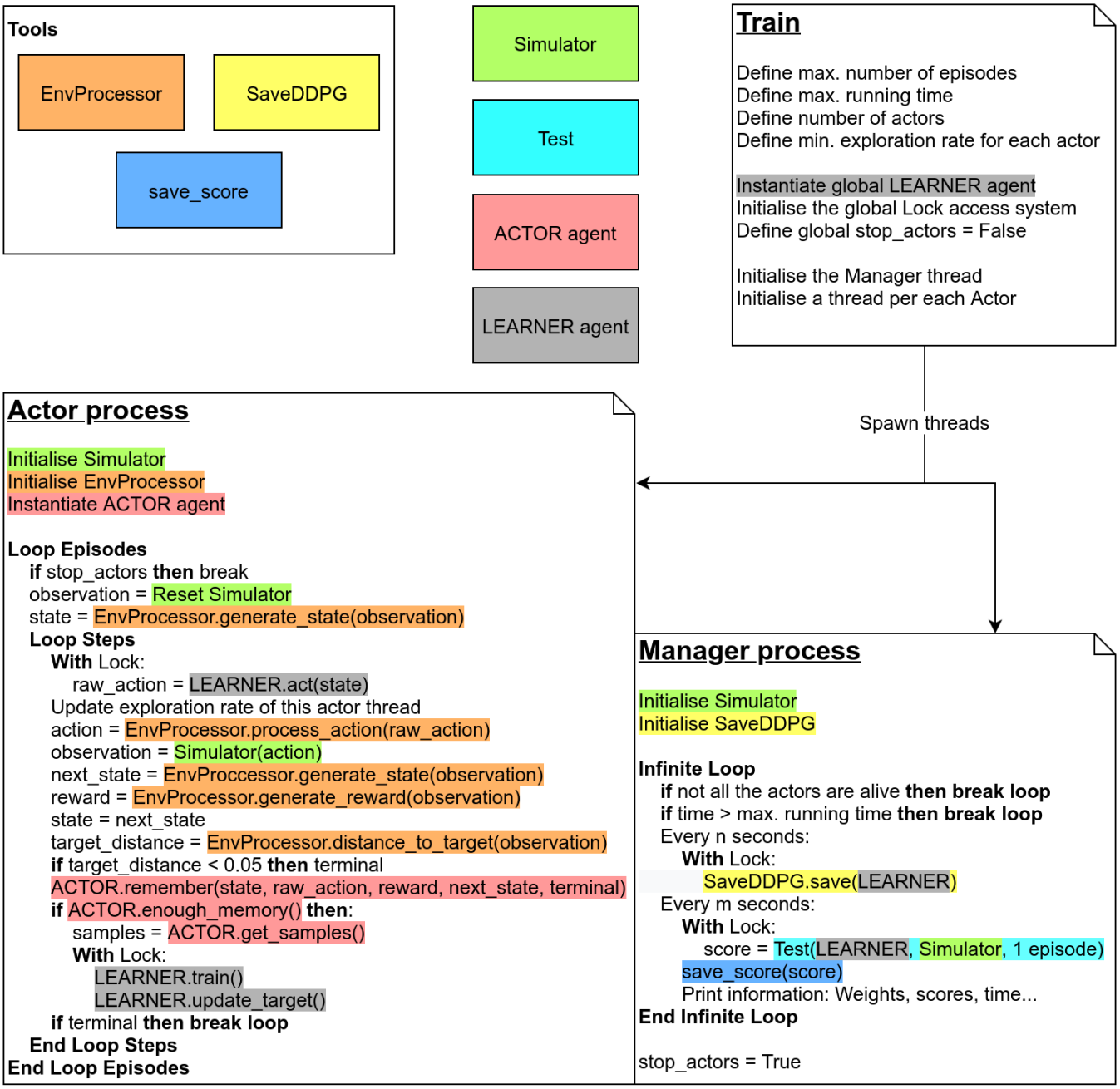


Figure 4.16.: Shared memory learning

5. Results

This chapter explains how the experiments have been designed and which parameters have been used. Afterwards the obtained results are shown and compared among the algorithms.

5.1. Parameters definition

After the implementation of the architectures they have been tested in order to see their performance and to compare their results. Before testing them the RL hyper parameters (section 3.1) must be defined. A deep research on the hyper parameters adjustment is out of the scope of this work, but anyway some of them are investigated to make the algorithms have a better performance. The fixed parameters, the ones not investigated, have been extracted from the paper *Continuous control with deep reinforcement learning*.[7] and *Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research*.[13], some other fixed parameters have been defined after simple tests and have not been further investigated.

Fixed predefined hyper parameters:

- Maximum memory capacity $C = 10^6$
- Initial exploration rate $\epsilon = 1.0$
- Exploration rate decay $\delta = 0.9995$
- Learning rates $\eta_{critic}, \eta_{actor} = 10^{-4}$
- Target update rate $\tau = 10^{-3}$

Investigated hyper parameters:

- Minimum exploration rate $\epsilon_{min} \in [0, \epsilon] \subset \mathbb{R}$
- Mini batch size $N \in [1, C] \subset \mathbb{N}$
- Discount rate $\gamma \in [0, 1] \subset \mathbb{R}$

Due to the lack of time, these parameters have been tested using only two different values per each one. These way there is a combination of $2^3 = 8$ possibilities. The Table 5.1 shows all the possible combinations with the values chosen for each of the parameters. These values have been decided based on the papers mentioned before and some experience acquired during this work.

Exploration rate ϵ	**Mini Batch N**	**Discount rate γ**
0.1	64	0.95
0.1	64	0.99
0.1	256	0.95
0.1	256	0.99
0.3	64	0.95
0.3	64	0.99
0.3	256	0.95
0.3	256	0.99

Table 5.1.: Parameters combinations

These eight possibilities have been tested using the task "FetchPickAndPlace-v1" with the DDPG for at least forty-thousand seconds each one. The Figure 5.1 shows the results obtained averaged over one-thousand measurements, to make the curves very smooth to be analysed, and using the number of episodes as the X axis.

There are four combination of hyper parameters that perform better, the ones with the discount rate $\gamma = 0.95$. Then the ones with a discount rate $\gamma = 0.99$ and a small mini batch $N = 64$ perform worse. Finally, the combination of a big mini batch $N = 256$ and a discount rate $\gamma = 0.99$ results into a very poor performance, even preventing the learning. Keeping the discount rate at 0.95 seems that the other two parameters do not matter significantly. But if the data is expressed with respect to the time instead of episodes, as in the Figure 5.2, the effects of the mini batch and exploration rate are more visible. In this new representation, it is possible to see that a big mini batch of $N = 256$ results into a much slower learning, due to an increase of the computations. So among the first four best solutions, only the ones with a mini batch of $N = 64$ perform much better than the others. Therefore, the best combination of the studied parameters seems to be an exploration rate $\epsilon_{min} = 10\% - 30\%$, a mini batch $N = 64$ and a discount rate $\gamma = 0.95$. It looks like the exploration rate of $\epsilon_{min} = 10\%$ performs slightly better than the other one, so this will be picked as the best when defining the parameter values for the further tests (Table 5.2).

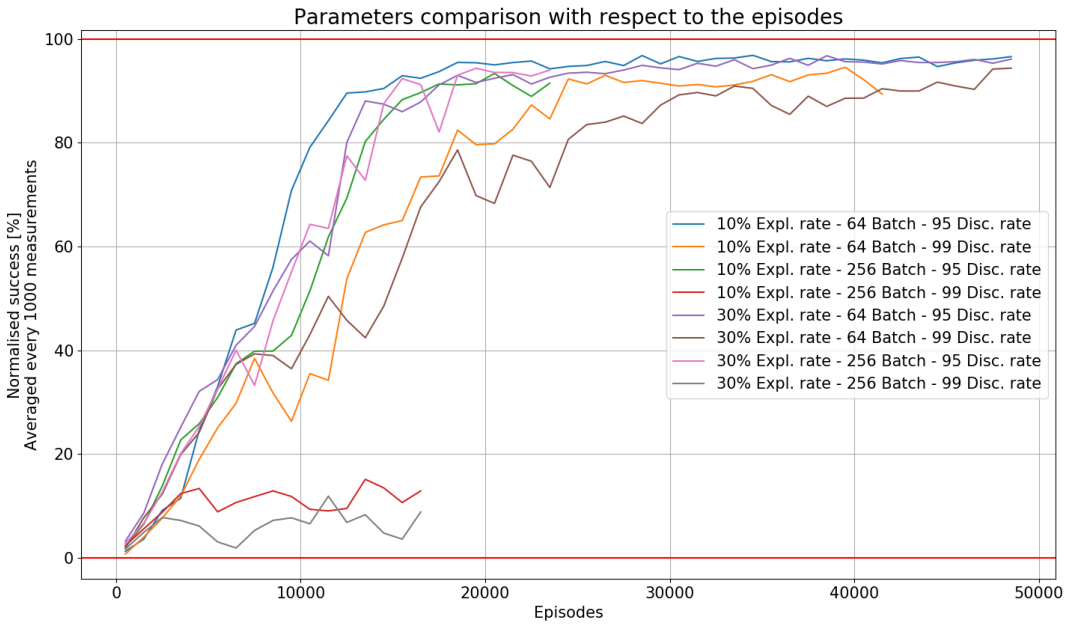


Figure 5.1.: Parameters with respect to episodes

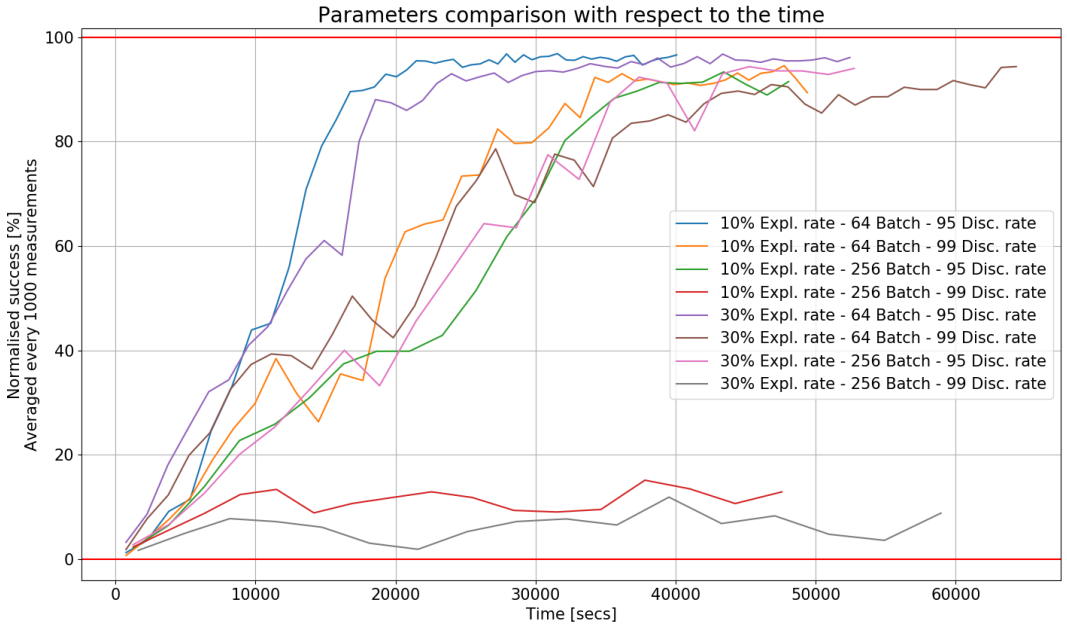


Figure 5.2.: Parameters with respect to time

5.2. Experiments definition

The Table 5.2 shows the hyper parameters used for the experiments, researched in the previous section. These are the parameters used for all the algorithms, the DDPG and the distributed ones. The only parameter which changes is the minimum exploration rate ϵ_{min}, because in the distributed algorithms each of the actors has a different value in order to enhance its exploration.

Parameter	Symbol	Value
Maximum memory capacity	C	10^6
Initial exploration rate	ϵ	1.0
Minimum exploration rates	ϵ_{min}	0.1
Exploration rate decay	δ	0.9995
Mini batch size	N	64
Discount rate	γ	0.95
Learning rates	η	10^{-4}
Target update rate	τ	10^{-3}

Table 5.2.: Parameters comparison

All the experiments have been run in a computer, provided by the university, with the following characteristics:

4 cores Intel(R) Core(TM) i5-6600 CPU @ 3.30GHz
RAM 16GB

The training time has been defined as 45000 seconds, which is 12.5 hours. After some previous testing, this has shown to be a range of time long enough to show some results even for the worst performing architectures.

The Table 5.3 shows all the performed experiments. The number of actors has been chosen based on the microprocessor architecture of the computer used. Each actor takes one process plus the train program and the learner or manager, so the total number of processes is the number of actors plus two. Since the computer has only four cores and reaching the maximum core capacity is not wanted, a maximum of four actors has been considered a good option. That is why the experiments have been done with a maximum number of four actors. The distributed algorithms have also been tested with only one actor, because this way it is possible to compare the effect of the architecture in the performance. Due to the lack of time only one experiment more per algorithm has

been done. Between the maximum number of actors (four) and the minimum (one), two actors has been chosen as the third experiment to conduct.

As said before, all the parameters used are defined in the Table 5.2 but the minimum exploration rates, which are changed if there are multiple actors. If the algorithm is the DDPG or is one of the distributed algorithms with only one actor, then the minimum exploration rate is defined as 10%. But when there are multiple actors a range of values between 10% and 30% is chosen. The range is up to 30% because as seen in the previous section, the performance with a minimum exploration rate of 30% is practically the same as the 10% one.

	Actors	**Exploration rate ϵ**
DDPG	1	0.10
Distributed with centralised learning	1	0.10
	2	0.10, 0.20
	4	0.10, 0.15, 0.20, 0.25
Distributed with decentralised learning	1	0.10
	2	0.10, 0.20
	4	0.10, 0.15, 0.20, 0.25
Distributed with shared memory	1	0.10
	2	0.10, 0.20
	4	0.10, 0.15, 0.20, 0.25

Table 5.3.: Experiments

In order to keep track of the learning process, all the algorithms have been tested and the networks weights periodically saved during the training. The frequency of the testing and saving has an effect on the performance, since the training must be stopped to test the algorithm, therefore there must be a balance between getting enough data and interfering as few as possible the learning. Constraining to these requirements, all the algorithms are tested every ca. ten seconds and the weighs are saved every ca. sixty seconds, after some previous experiments this has shown a good balance.

A part from the learning curves, at the end of the training each of the agents has been tested. Once the agents have been trained, the probability distribution of the results does not follow a Gaussian shape, since there is a much higher probability to have a result closer to 100% than 0%. To compare the results of the different algorithms not only the results average is wanted but also the variance to asses the repeatability. Therefore

to have a Gaussian distribution would be ideal. In order to get a Gaussian distribution the central limit theorem can be used. If the agent is tested n number of episodes then the scores obtained will not follow a Gaussian. But if these n scores are averaged and the test is done m times, then the distribution of the m averaged scores will follow a Gaussian, due to the central limit theorem. Based on this idea all the trained agents have been tested one hundred times one hundred episodes to obtain one hundred averages. This way an average and a variance can be extracted.

Experiments with delay

After some previous tests, it has been seen that the architecture of the algorithms, which simulate to be in different physical machines, has a big impact in the performance, if it is compared with the DDPG and the distributed algorithm with shared memory space. Besides, these algorithms are intended to be applied in real robots, therefore the velocity of the simulator does not fit the real velocity of a moving robot, since the simulator can run much faster.

In order to prove, that the architecture effect would be negligible in a real robot, additional experiments have been defined with a small delay after each action. This delay simulates the time a robot would take before reaching the next step. The Table 5.4 shows the conducted experiments with the delay. The DDPG has been tested with the delay to use it as a benchmark for the comparison. Because of the delay, these tests take much longer than the other ones, therefore due to the lack of time the experiment with two actors has not been done.

The delay after each action has been defined as 0.3 seconds, considered a big enough value to simulate physical movements. The experiments have been run for a period of 100000 seconds, which is 27,78 hours.

	Actors	**Exploration rate ϵ**
DDPG	1	0.10
Distributed with centralised learning	1	0.10
	4	0.10, 0.15, 0.20, 0.25
Distributed with decentralised learning	1	0.10
	4	0.10, 0.15, 0.20, 0.25

Table 5.4.: Experiments with delay

5.3. Experiments results

The following subsections show the results obtained in each of the defined experiments.

5.3.1. Deep Deterministic Policy Gradient

The Figure 5.3 shows the learning progress of the DDPG algorithm under the conditions defined in the previous section. The results have been averaged every fifty measurements in order to smooth the curves. As seen, the main part of the learning takes place the first fifteen-thousand seconds, after that the learning stabilises and becomes stationary with small variations.

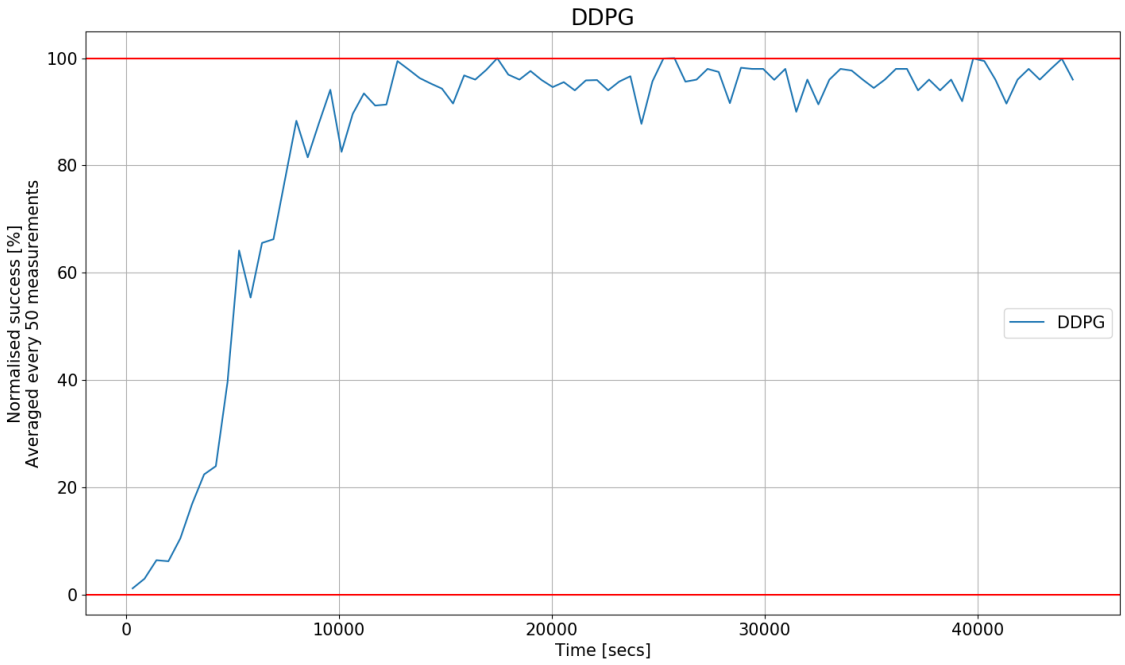


Figure 5.3.: Deep deterministic policy gradient

The Table 5.5 shows the tests results once the training time has finished. As seen, it reaches a quite high value but still with a failure ratio of ca. five per cent.

	Average	Standard deviation
DDPG	95.43%	1.97

Table 5.5.: DDPG test results

5.3.2. Distributed architecture with centralised learning

The Figure 5.4 shows the learning progress of the distributed architecture with centralised learning.

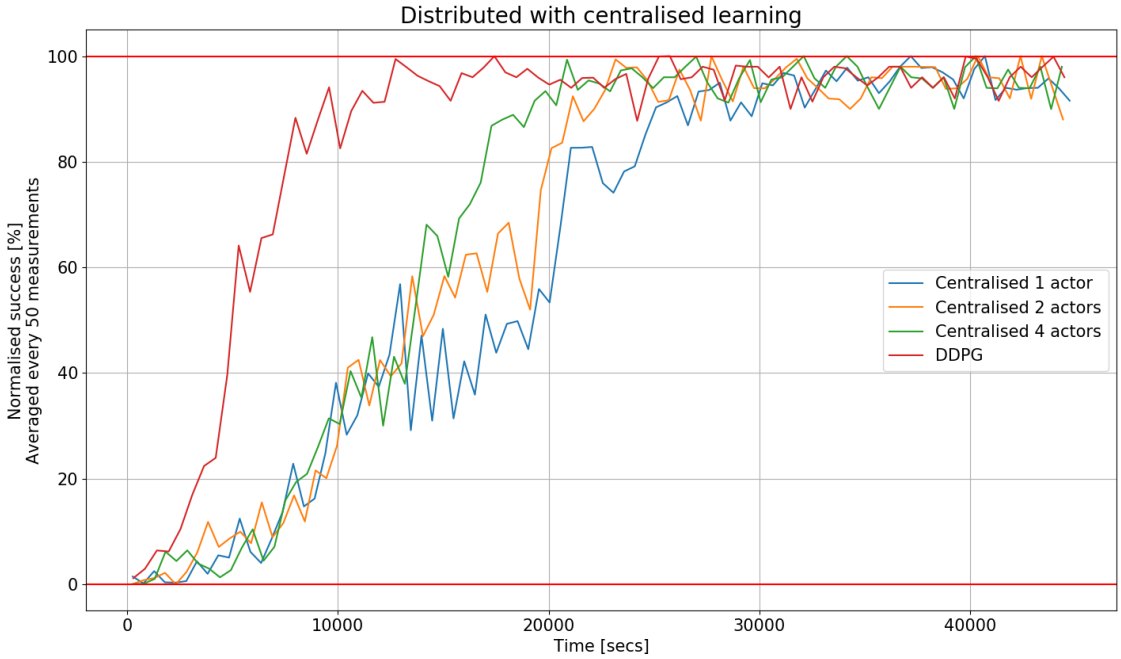


Figure 5.4.: Distributed architecture with centralised learning

It can be seen that the performance is worse than with the DDPG. In the distributed architecture with one actor the stationary part is reached after ca. thirty-thousand episodes, so fifteen-thousand seconds more than with the DDPG. This is caused by the distributed architecture, sending the gradients and the weights has a cost in the learning performance. Now, if the performance of the architecture with two and four actors is compared with the results with one actor, the results are much better. The learning progress seems to be improved by the number of actors, so the more actors the faster learning. These results are still behind the DDPG performance but show what was expected, an improvement if more actors are used.

The Table 5.6 shows the tests statistics after the training. Statistically these averaged scores with these standard deviations can be considered the same. Therefore in this case the number of actors does not look to have an effect on the maximum score.

	Average	Standard deviation
1 actor	95.62%	2.02
2 actors	96.20%	1.81
4 actors	96.16%	1.67

Table 5.6.: Centralised learning test results

5.3.3. Distributed architecture with decentralised learning

The Figure 5.5 pictures the learning process of this architecture, showing only the results of the first actor in the cases there are many.

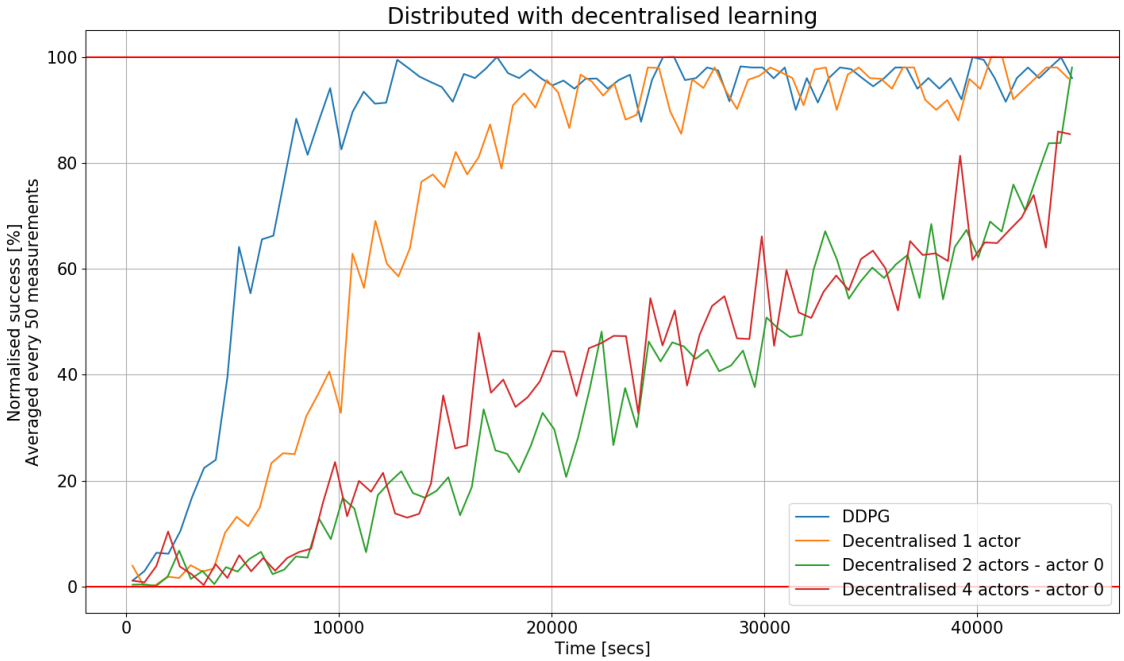


Figure 5.5.: Distributed architecture with decentralised learning

Like in the previous case, here the architecture has a big impact. In this case, with one actor the performance is much better than with multiple actors, even better than the performance with one actor in the distributed architecture with centralised learning. This is because here with one actor there is only one pair of networks, the ones the actor has. On the other hand, in the previous architecture, there is also a learner so there are two pairs of networks, which can diverge making the algorithm slower. Having said

that, here the performance with more than one actor decreases drastically. As it may be seen, the results with two and four actors are much worse than with one. This is caused, because with one actor the weights averaging does not have any effect and the actor receives its same weights, there is no risk of divergence. With more than one actor, all the actors receive averaged weights producing a slow down, because the actors with bad performance drag the ones with a better one. The same way it happens with the gradients. The cost of avoiding the divergence is high. In addition, four actors do not look like to have a better performance than two. An explanation for that could be that the effect of the architecture, to avoid the divergences, grows with the number of actors, so at the end, the advantages of having more actors are undermined proportionally by the disadvantages produced by the architecture effects.

As expected, the Table 5.7 shows that the experiments with multiple actors have a much worse performance. If they had been trained longer the would have reached the same results as the architecture working with one actor, but with the forty-five-thousand seconds they haven't had enough time to reach the stationary stage.

	Average	**Standard deviation**
1 actor	92.79%	2.59
2 actors - actor 0	82.21%	3.23
4 actors - actor 0	84.51%	3.01

Table 5.7.: Decentralised learning test results

The full plots and tables with all the actors can be found in the section A.1. They have not been added here because they are almost identical to their tween actors and do not add any relevant information.

5.3.4. Distributed architecture with shared memory space

The Figure 5.6 shows the learning progress of the distributed architecture with shared memory space. As it can be seen the results are the same as the ones obtained with the DDPG. Besides, there is no difference in the results with the different number of actors. This is produced by the implementation and not by the architecture. As mentioned in the section 4.9, this architecture has been implemented using threads instead of processes. Threads in Python run only in one processor, so it is not a real parallelising. This produces that at the end the performance is the same regardless of the number of actors.

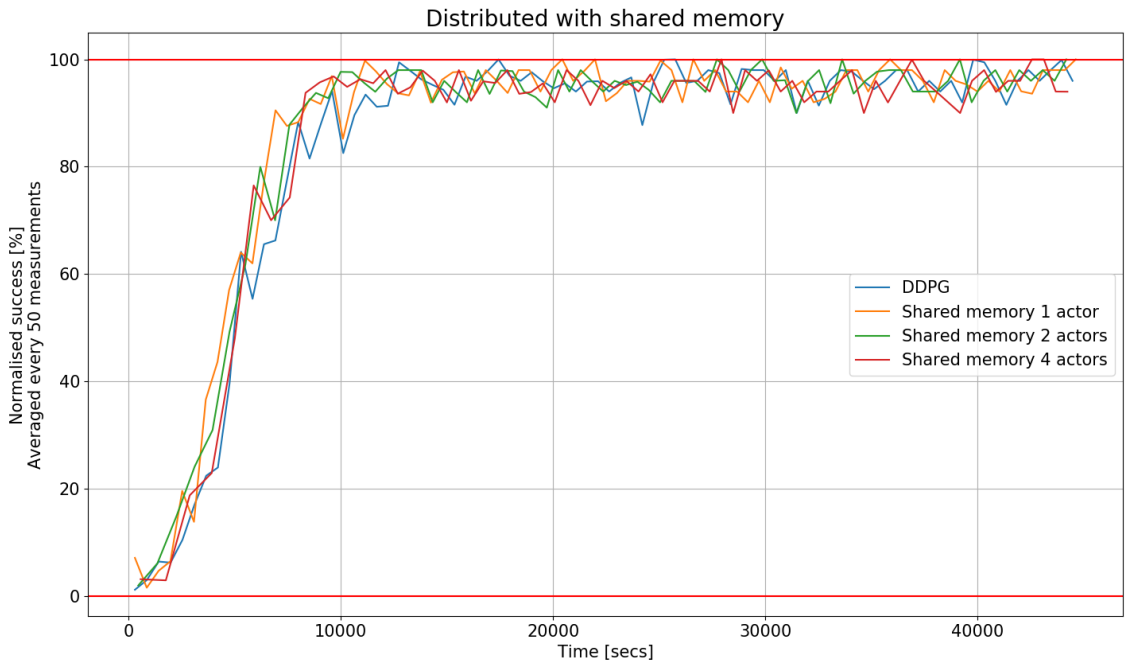


Figure 5.6.: Distributed architecture with shared memory space

The Table 5.8 shows the tests statistics. As expected, statistically all of them can be considered the same.

	Average	Standard deviation
1 actor	96.04%	2.09
2 actors	96.14%	2.08
4 actors	94.59%	2.25

Table 5.8.: Shared memory learning test results

5.3.5. Deep Deterministic Policy Gradient with delay

The Figure 5.7 shows the learning progress of the DDPG with a delay in comparison with the same algorithm without the delay. Obviously it is much slower than without the delay. Even with one-hundred-thousand seconds it is not able to reach the stationary stage, only being able to reach a score of ca. seventy percent. This results with delay will be the ones used to see if the effect of the architectures in the distributed algorithms becomes negligible when it is applied in a much slower environment.

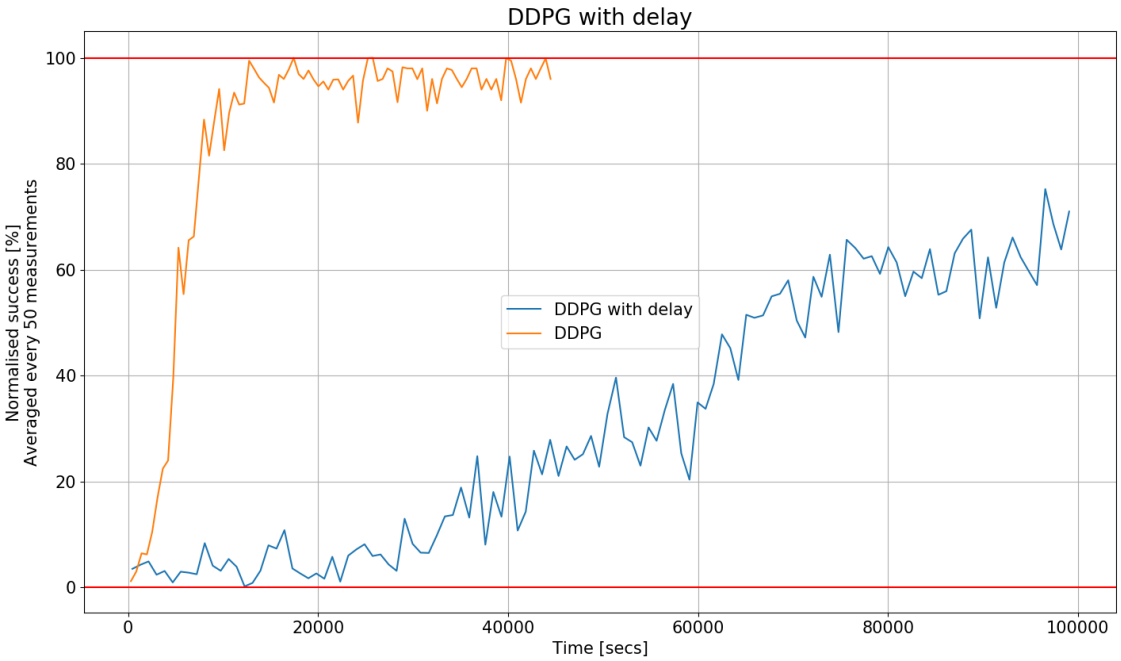


Figure 5.7.: Deep Deterministic Policy Gradient with delay

The Table 5.9 shows the tests results once the training time has finished.

	Average	Standard deviation
DDPG with delay	73.06%	3.32

Table 5.9.: DDPG with delay test results

5.3.6. Distributed architecture with centralised learning and delay

The Figure 5.8 proves the hypothesis formulated before. When the distributed architecture with a centralised learning is applied to a slower environment the effect of the architecture becomes negligible. As expected with one actor the results are almost identical as the DDPG ones with delay, and when multiple actors are used the learning is improved substantially. Therefore, this algorithm could be applied to real robots to learn in parallel improving their performance.

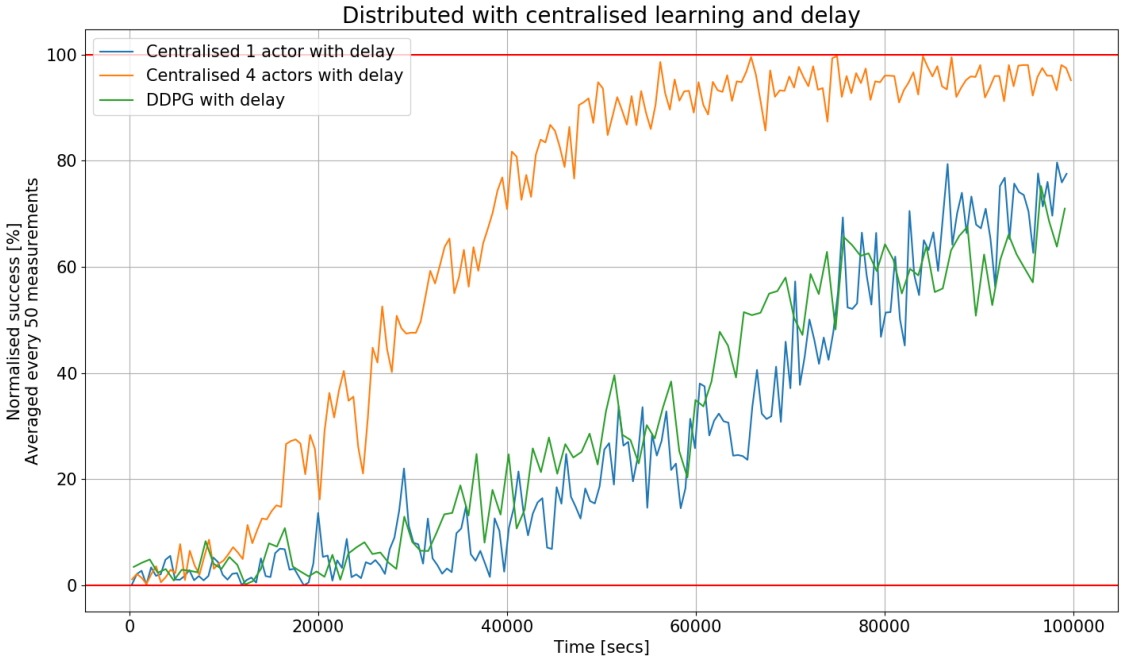


Figure 5.8.: Distributed architecture with centralised learning and delay

The Table 5.10 shows the tests results. As expected, the test with four actors reaches the maximum score.

	Average	Standard deviation
1 actor	79.97%	3.48
4 actors	95.79%	2.19

Table 5.10.: Centralised learning with delay test results

5.3.7. Distributed architecture with decentralised learning and delay

The Figure 5.9 shows the results when the delay is applied to the distributed architecture with decentralised learning. Contrary to the centralised learning architecture, here looks like the delay does not improve the performance, reducing the impact of the architecture, with respect to the DDPG. As it may be seen, if multiple actors are used does not produce any improvement either. Further experimentation is required to determinate why this improvement does not take place. Other delay times should be tested and the experiments should be run longer to get more information.

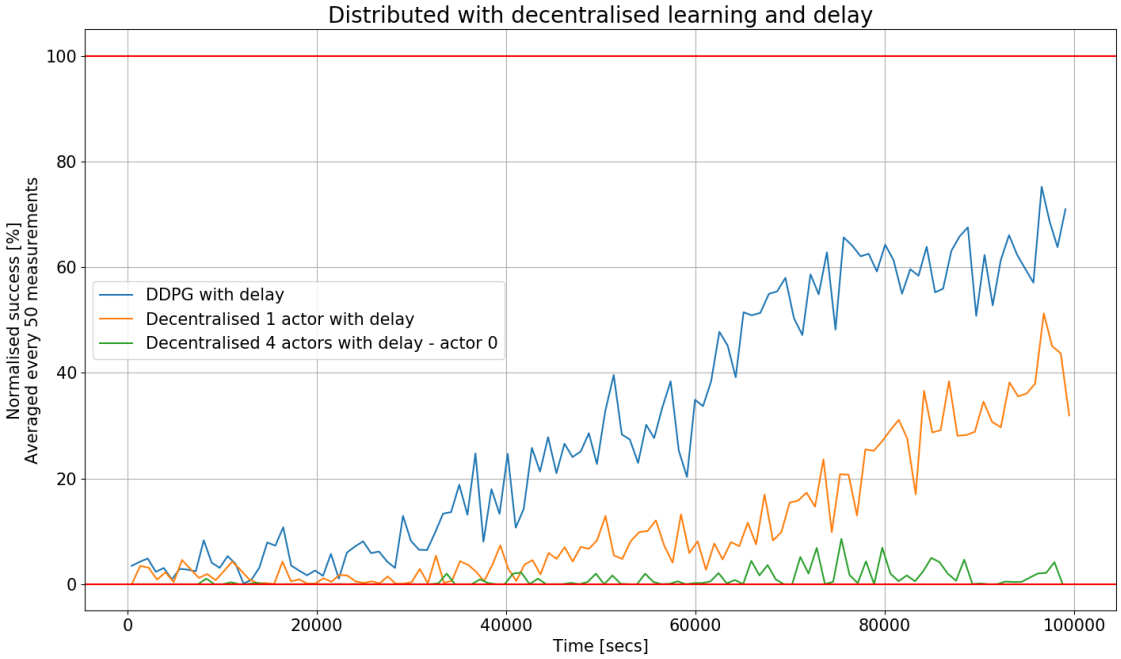


Figure 5.9.: Distributed architecture with decentralised learning and delay

The Table 5.11 shows the test results. As awaited, the results are very poor.

	Average	Standard deviation
1 actor	47.02%	4.03
4 actors - actor 0	02.85%	1.20

Table 5.11.: Decentralised learning with delay test results

The full plots and tables with all the actors can be found in the section A.2.

5.4. Architectures comparison

The Figure 5.10 shows the learning progress of all the algorithms in the same plot in order to easy the comparison.

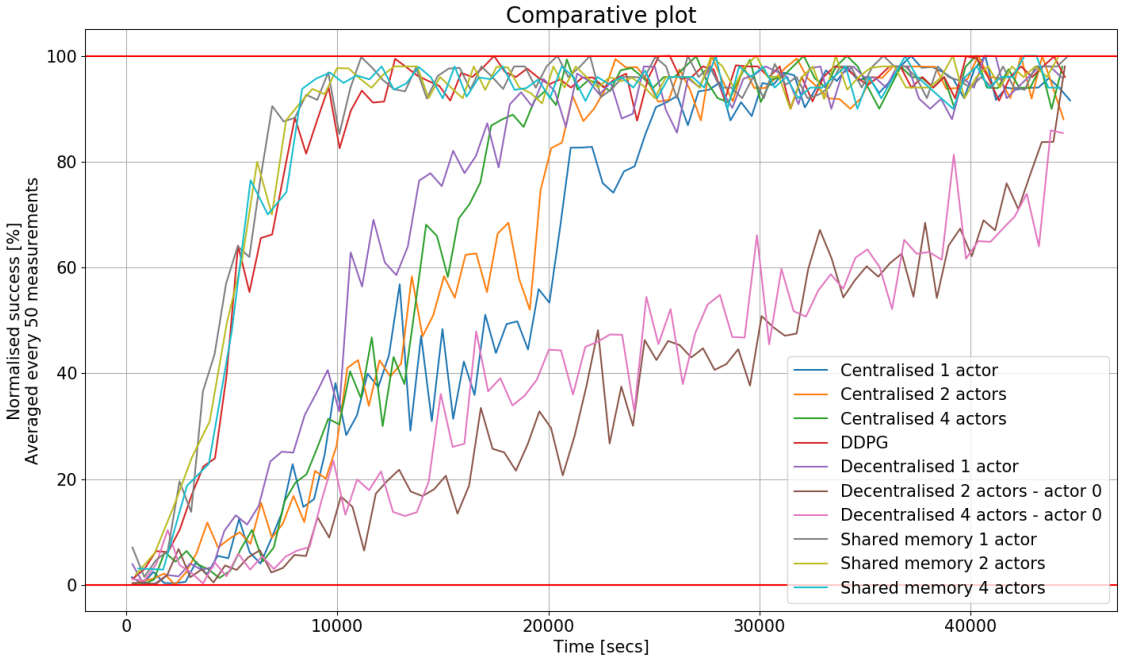


Figure 5.10.: Comparative plot of all the implemented architectures

As seen in the plot, the best performance is obtained with the non distributed architecture DDPG and the Distributed architecture with shared memory space. Theoretically this distributed architecture should have much better results than the DDPG but due to the implementation this can not be seen in these results. On the other hand, the other two distributed architectures show a poorer performance due to the undermining effect of the communication between actors and learners. If the experiments with one actor are not taken in account, because there is no point to use them in a real application, it is possible to see that the distributed architecture with a centralised learning performs much better than the one with a decentralised learning. As said before, this can be caused by the more complex architecture used in the decentralised learning algorithm. A part from the learning speed, all the algorithms show the same results once they have reached the stationary stage, this may indicate that the distributed architectures do not have a big impact in the precision. This is still to be double proved with further experiments.

The Figure 5.11 shows the learning progress for all the architectures tested with a delay. It can be seen that in this case the results change totally for the distributed algorithm with centralised learning with respect to the DDPG. The centralised learning algorithm with multiple actors, in this case four, shows a much better performance than the basic DDPG with the delay. If this is compared with the performance of the distributed architecture with decentralised learning, the difference is huge. The effect of the delay does not overcome the strong effect of the architecture and the results with one or multiple actors are still much worse than the ones of the DDPG with delay.

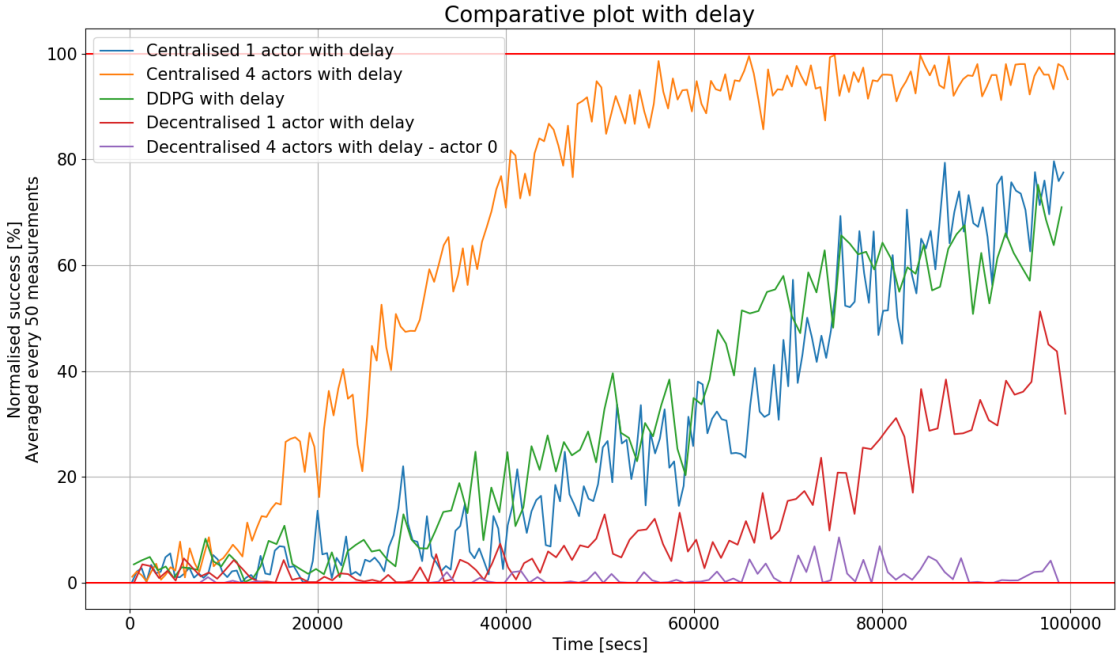


Figure 5.11.: Comparative plot of all the implemented architectures with delay

6. Conclusions

This project has shown the development and implementation of three different distributed RL architectures with their learning based on the DDPG algorithm. They have been designed based on recent advances on the field and tested using a simulated robotic arm environment. The distributed architectures with centralised learning, with and without shared memory space, are a particular version of already existing architectures, while the distributed architecture with decentralised learning has been an attempt to explore something new. After all of this work some conclusions may be drawn.

The learning distribution does not guarantee a better performance than the non distribution. How the architecture is designed and implemented has a big impact on the stability and on the learning performance. This has been seen in the experimental results obtained with the architectures which simulate different physical robots. The architecture impact in both cases has shown, to undermine totally the advantages of the distribution.

The degree of the architecture impact on the performance, changes depending on the complexity of the algorithm and how the learning has been distributed. This has been seen in the distributed architecture with decentralised learning. The impact of this architecture on the performance has shown to be much higher than in the distributed architecture with centralised learning. The methods used to avoid the divergence of the actors make the architecture more complex and slow which results in a much higher impact.

It is important to implement the architectures using real multiprocessing, if the actors do not work in parallel all the advantages of the distribution may be lost. This problem has occurred in this work with the distributed architecture with shared memory space. The implementation using the threading Python module does not parallelise the learning physically through different computer processors, and this has caused the lost of all the distribution advantages.

The simulated environments can run much faster than real robots, at these speeds the effect of the architectures has a stronger impact and can undermine totally the

advantages of the distribution. If a much slower environment is used, simulating a real robot, the impact of the architectures may be highly reduced making the distribution to increase the performance surpassing the non distributed architectures. This has been proved by the experiments made with the distributed architecture with centralised learning. It has been possible to see that if a delay is applied after each action, simulating a slower environment, the effect of the architecture is drastically reduced with respect to the performance, making the learning much faster than with the non distributed architecture.

Finally, it is important to say, that this work has been a small introduction to the matter and there is still a lot of work and research to be done to solve all the problems found during the process of making this project.

7. Future work

During this work, many ideas have brought up, but due to the lack of time and being out of the initial scope they have neither been researched nor implemented. Here they are summarised as a reference for future work.

With regard to the design of the distributed architectures developed, would be interesting to investigate other ways to manage the weights and the gradients sent. For example, empting the gradients containers when a new gradient is received. Also related with the architectures, would be worthwhile to define a parameter to regulate the frequency of the weights updating in the decentralised architecture. If they are updated less frequently then the architecture effect on the performance might be reduced.

Concerning the implementation of the algorithms, the shared memory architecture could be built using real multiprocessing to see its real potential. Besides this, it would be interesting to try other neural networks structures to see if the precision of the agents can be enhanced.

Respecting the experimentation and testing, a systematic methodology, such as a factorial design or a genetic algorithm, could be used in order to find the proper learning hyper parameters. Also interesting would be that, all the distributed architectures were tested with more than four actors and for a longer time, specially the experiments with a delay. Furthermore, different delays should be tested or even a variable delay which varied with respect to the movements increments to make it more realistic. Moreover, different exploration rates could be tested than the ones used in the distributed algorithms to research their impact in the performance.

Once these previous possibilities have been investigated, it would be of interest to test the algorithms in real robots and see if the performance predictions apply.

Going further, new unexplored branches of the classification tree (Figure 2.2) could be explored. Also, all the architectures could be redesigned based on DQN, to be tested on discrete action spaces. Finally, many other features could be added to the algorithms in order to improve its performance, such as a prioritised experience replay [16].

A. Distributed architecture with decentralised learning additional results

This annex contains the plots of the distributed architecture with decentralised learning with all the actors.

A.1. Experiments without delay

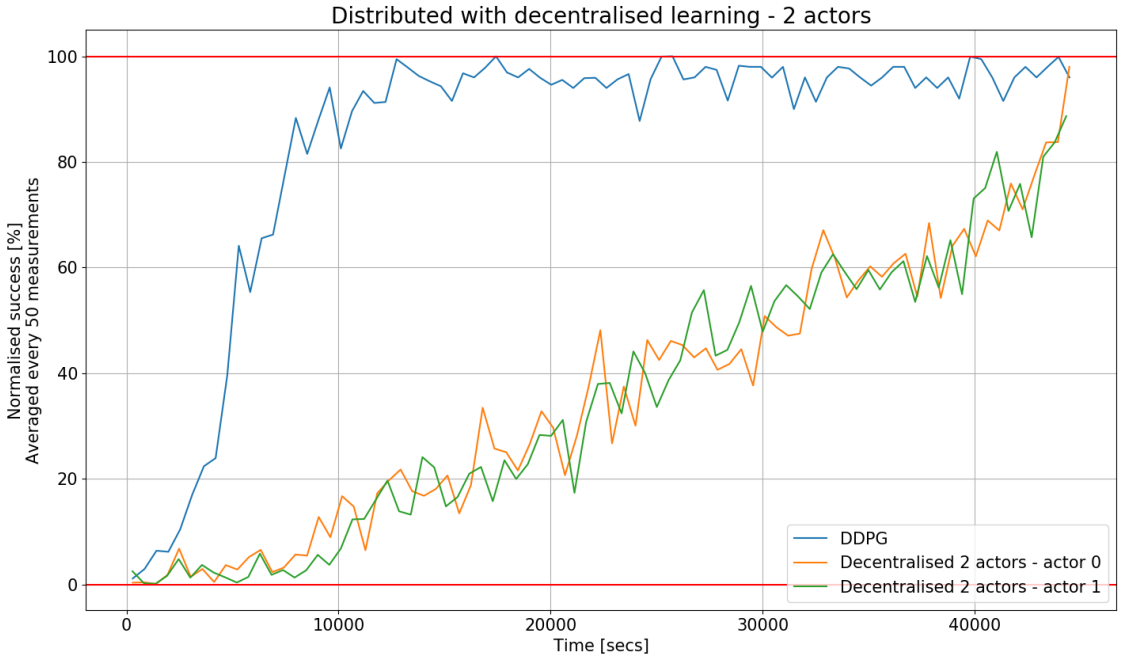


Figure A.1.: Distributed architecture with decentralised learning - 2 actors

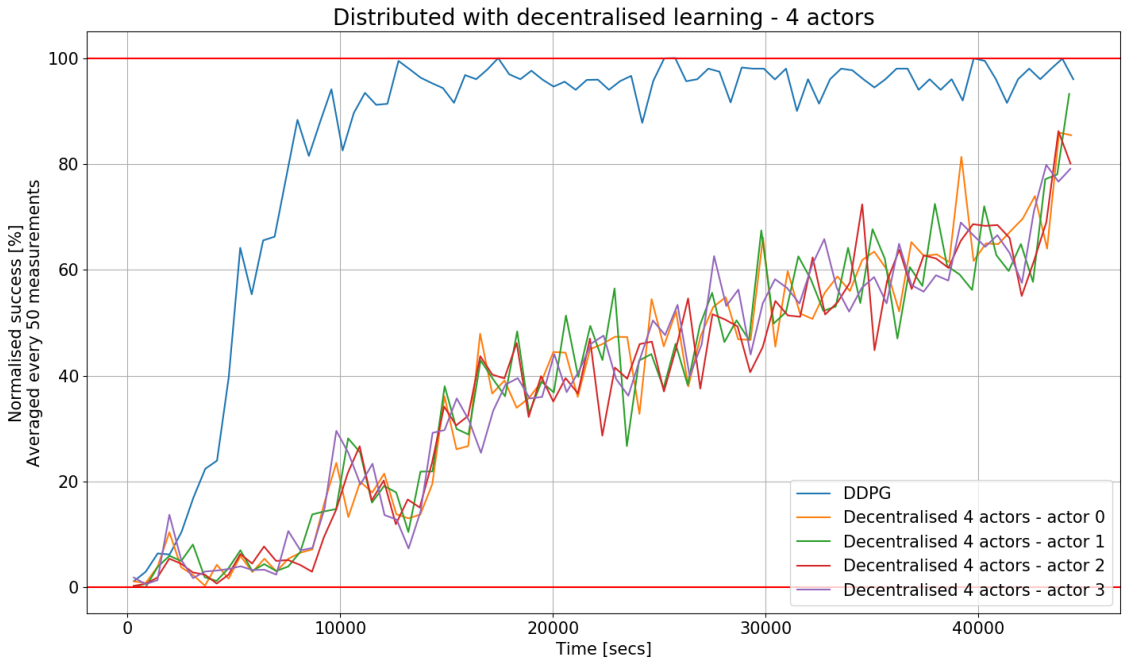


Figure A.2.: Distributed architecture with decentralised learning - 4 actors

	Average	Standard deviation
1 actor	92.79%	2.59
2 actors - actor 0	82.21%	3.23
2 actors - actor 1	85.08%	3.38
4 actors - actor 0	84.51%	3.01
4 actors - actor 1	85.11%	3.10
4 actors - actor 2	84.65%	2.95
4 actors - actor 3	85.72%	3.13

Table A.1.: Decentralised learning test results with all the actors

A.2. Experiments with delay

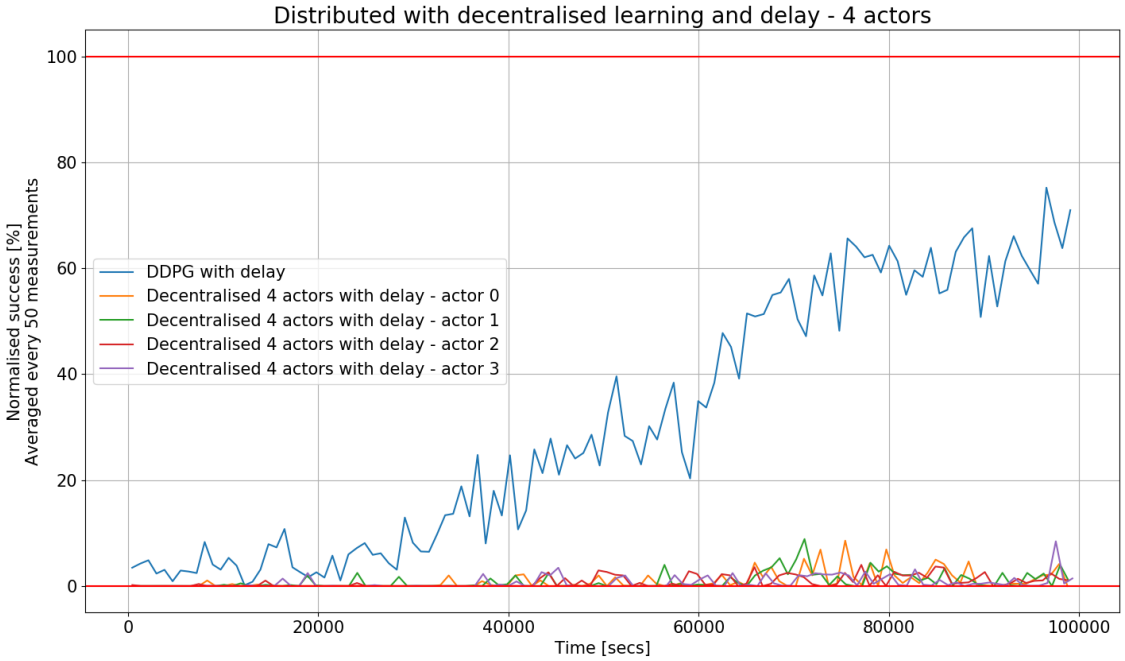


Figure A.3.: Distributed architecture with decentralised learning and delay - 4 actors

	Average	Standard deviation
1 actor	47.02%	4.03
4 actors - actor 0	02.85%	1.20
4 actors - actor 1	02.90%	1.35
4 actors - actor 2	02.38%	1.01
4 actors - actor 3	02.79%	1.37

Table A.2.: Decentralised learning test results with all the actors and delay

B. Programs usage

This annex describes how to run the programs implemented in the project.

B.1. Training

The training program is started with the following command for all the algorithms:

```
python train.py
```

The parameters are different depending on the algorithms.

For the DDPG:

-env, - -environment	String with the environment name. Default = "FetchPickAndPlace-v1".
-e, - -episodes	Integer with the maximum number of episodes to run. Default = 1000000
-t, - -max_time	Integer with the maximum time to run. Default = 45000secs
-d, - -delay	If given, the delay will be activated.

For all the other algorithms:

-env, - -environment	String with the environment name. Default = "FetchPickAndPlace-v1".
-t, - -max_time	Integer with the maximum time to run. Default = 45000secs
-a, - -actors	Integer with the number of actors.
-d, - -delay	If given, the delay will be activated.
-expl, - -exploration	String with the shape of a python list with the minimum exploration rate per each actor. e.g. "[0.1,0.2]"

B.2. Testing

To test the DDPG, distributed centralised and decentralised algorithms results the following command has to be run with the following parameters:

```
python tester.py
```

-p, - -path	String with the path of the .pickle file. If not given, a filedialog will appear to select the file.
-env, - -environment	String with the environment name. Default = "FetchPickAndPlace-v1".
-e, - -episodes	Integer with the number of episodes to run.

The distributed algorithm with shared memory space uses a different file system to save the training, therefore the testing is slightly different. The parameters are all the same, but there is a new one that must be added:

-c, - -continuous	If given, the program knows that a shared memory space learning is tested.

In this case, the files do not have a .pickle format, therefore the **All files *.*** option must be picked in the file dialog, and any of the files generated by the trainer can be selected, since the tester just needs to know the folder where they are.

B.3. Plotting

To plot the results the following command has to be run with the following parameters:

```
python plotter.py
```

-p, - -path	String with the path of the text file with the data to plot. If not given, a filedialog will appear to select the files. Multiple files can only be opened using the filedialog.
-m, - -measurements	Integer with the number of measurements to average. Default = 1.
-i, - -info	String with the title of the plot.
-e, - -episodes	If given, X axis will be episodes. If not given, X axis will be time.

Bibliography

[1] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. P. Lillicrap. "Distributed Distributional Deterministic Policy Gradients." In: *ICLR (Poster)*. OpenReview.net, 2018.

[2] M. Breyer, F. Furrer, T. Novkovic, R. Siegwart, and J. I. Nieto. "Comparing Task Simplifications to Learn Closed-Loop Object Picking Using Deep Reinforcement Learning." In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 1549–1556.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[4] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu. *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. cite arxiv:1802.01561. 2018.

[5] Keon. *deep-q-learning*. Code: `https://github.com/keon/deep-q-learning`. 2018. Commit: c99b677b1aa0024f3809981cb5e8252b9abf4946

[6] M. Kweon. *A3C-Tensorflow*. Code: `https://github.com/kkweon/A3C-Tensorflow`. 2017. Commit: 64606f1ee72f648b3f7aa15e0902fdf4da7b377a

[7] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. "Continuous control with deep reinforcement learning." In: *ICLR*. (ICLR). Ed. by Y. Bengio and Y. LeCun. 2016.

[8] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning*. cite arxiv:1602.01783. 2016.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing atari with deep reinforcement learning." In: *arXiv preprint arXiv:1312.5602* (2013).

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836.

[11] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. "Massively Parallel Methods for Deep Reinforcement Learning." In: *CoRR* abs/1507.04296 (2015).

[12] piotrplata. *keras-ddpg*. Code: `https://github.com/piotrplata/keras-ddpg`. 2018. Commit: 8b7022046eb7de28018dac71bbef7ad8a962b23a

[13] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba. "Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research." In: *CoRR* abs/1802.09464 (2018).

[14] I. Popov, N. Heess, T. P. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerík, T. Lampe, Y. Tassa, T. Erez, and M. A. Riedmiller. "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation." In: *CoRR* abs/1704.03073 (2017).

[15] G. Sartoretti, Y. Wu, W. Paivine, T. K. S. Kumar, S. Koenig, and H. Choset. "Distributed Reinforcement Learning for Multi-robot Decentralized Collective Construction." In: *DARS*. Ed. by N. Correll, M. Schwager, and M. W. Otte. Vol. 9. Springer Proceedings in Advanced Robotics. Springer, 2018, pp. 35–49. ISBN: 978-3-030-05816-6.

[16] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. "Prioritized Experience Replay." In: *CoRR* abs/1511.05952 (2015).

[17] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

[18] C. J. C. H. Watkins. "Learning from Delayed Rewards." PhD thesis. King's College, Oxford, 1989.

[19] Q. Yang, Y. Liu, T. Chen, and Y. Tong. "Federated Machine Learning: Concept and Applications." In: *CoRR* abs/1902.04885 (2019).

[20] H. H. Zhuo, W. Feng, Q. Xu, Q. Yang, and Y. Lin. "Federated Reinforcement Learning." In: *CoRR* abs/1901.08277 (2019).